Developer Guide

Amazon Lookout for Vision

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon Lookout for Vision: Developer Guide

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

..... ix

What is Amazon Lookout for Vision? 1

Key benefits 1

Are you a first-time Amazon Lookout for Vision end user? 1

Setting up Amazon Lookout for Vision 2

Step 1: Create an AWS account 2

Sign up for an AWS account 2

Create a user with administrative access 3

Step 2: Set up permissions 4

Setting console access with AWS managed policies 5

Setting Amazon S3 bucket permissions 5

Assigning permissions 6

Step 3: Create the console bucket 7

Creating the console bucket with the Amazon Lookout for Vision console 8

Creating the console bucket with Amazon S3 9

Console bucket settings 10

Step 4: Set up the AWS CLI and AWS SDKs 10

Install the AWS SDKS 10

Grant programmatic access 11

Set up SDK permissions 14

Call an Amazon Lookout for Vision operation 18

Step 5: (Optional) Using your own AWS KMS key 22

Understanding Amazon Lookout for Vision 24

Choose your model type 25

Image classification model 25

Image segmentation model 25

Create your model 27

Create a project 27

Create a dataset 27

Train your model 29

Evaluate your model 29

Use your model 30

Use your model on an edge device 30

Use your dashboard 30

Getting started **31**
Step 1: Create the manifest file and upload images 33
Step 2: Create the model 34
Step 3: Start the model 41
Step 4: Analyze an image 44
Step 5: Stop the model 49
Next steps 51
Creating your model **52**
Creating your project 52
Creating a project (console) 53
Creating a project (SDK) 53
Creating your dataset 55
Preparing images for a dataset 56
Creating the dataset 57
Local computer 59
Amazon S3 bucket 60
Manifest file 63
Labeling images 90
Choosing the model type 91
Classifying images (console) 91
Segmenting images (console) 92
Training your model 96
Training a model (console) 97
Training a model (SDK) 98
Troubleshooting model training 104
Anomaly label colors don't match color of anomalies in mask image 104
Mask images aren't in PNG format 106
Segmentation or classification labels are inaccurate or missing 107
Improving your model **109**
Step 1: Evaluate the performance of your model 109
Image classification metrics 109
Image segmentation model metrics 110
Precision 110
Recall 111
F1 score 111
Average Intersection over Union (IoU) 112

Testing results ... 113
Step 2: Improve your model ... 113
Viewing performance metrics ... 115
Viewing performance metrics (console) ... 115
Viewing performance metrics (SDK) ... 116
Verifying your model ... 120
Running a trial detection task ... 121
Verifying trial detection results ... 122
Correcting segmentation labels with the annotation tool ... 123
Running your model ... 125
Inference units ... 125
Managing throughput with inference units ... 126
Availability Zones ... 128
Starting your model ... 128
Starting your model (console) ... 129
Starting your model (SDK) ... 130
Stopping your model ... 135
Stopping your model (console) ... 135
Stopping your model (SDK) ... 136
Detecting anomalies in an image ... 141
Calling DetectAnomalies ... 141
Understanding the response from DetectAnomalies ... 145
Classification model ... 145
Segmentation model ... 146
Determining if an image is anomalous ... 148
Classification ... 148
Segmentation ... 150
Showing classification and segmentation information ... 155
Finding anomalies with an AWS Lambda function ... 170
Step 1: Create an AWS Lambda function (console) ... 170
Step 2: (Optional) Create a layer (console) ... 172
Step 3: Add Python code (console) ... 173
Step 4: Try your Lambda function ... 178
Using your model on an edge device ... 183
Deploying a model to a core device ... 185
Core device requirements ... 185

Tested devices, chip architectures, and operating systems 186
Core device memory and storage 187
Required software 187
Setting up your core device 189
Setting up your core device 189
Packaging your model 191
Package settings 191
Packaging your model (Console) 194
Packaging your model (SDK) 194
Getting information about model packaging jobs 198
Writing your client application component 200
Setting up your environment 201
Using a model 203
Creating the client application component 208
Deploying your components to a device 213
IAM permissions for deploying components 213
Deploying your components (console) 214
Deploying the components (SDK) 215
Lookout for Vision Edge Agent API Reference 217
Detecting anomalies with a model 217
Getting model information 217
Running a model 218
DetectAnomalies 218
DescribeModel 224
ListModels 226
StartModel 227
StopModel 229
ModelStatus 230
Using the dashboard 232
Managing your resources 235
Viewing your projects 235
Viewing your projects (console) 236
Viewing your projects (SDK) 236
Deleting a project 239
Deleting a project (console) 239
Deleting a project (SDK) 240

Viewing your datasets ... 242
Viewing the datasets in a project (console) ... 242
Viewing the datasets in a project (SDK) ... 242
Adding images to your dataset ... 245
Adding more images ... 245
Adding more images (SDK) ... 246
Removing images from your dataset ... 252
Removing images from a dataset (Console) ... 252
Removing images from a dataset (SDK) ... 253
Deleting a dataset ... 254
Deleting a dataset (console) ... 242
Deleting a dataset (SDK) ... 254
Exporting datasets from a project (SDK) ... 257
Viewing your models ... 265
Viewing your models (console) ... 266
Viewing your models (SDK) ... 266
Deleting a model ... 269
Deleting a model (console) ... 269
Deleting a model (SDK) ... 269
Tagging models ... 273
Tagging models (console) ... 273
Tagging models (SDK) ... 275
Viewing your trial detection tasks ... 276
Viewing your trial detection tasks (console) ... 277
Example code and datasets ... 278
Example code ... 278
Example datasets ... 278
Image segmentation datasets ... 279
Image classification dataset ... 279
Security ... 282
Data protection ... 282
Data encryption ... 283
Internetwork traffic privacy ... 285
Identity and access management ... 285
Audience ... 285
Authenticating with identities ... 286

Managing access using policies 289
How Amazon Lookout for Vision works with IAM 292
Identity-based policy examples 298
AWS managed policies 302
Troubleshooting 312
Compliance validation 313
Resilience 315
Infrastructure security 315
Monitoring 316
Monitoring with CloudWatch 316
CloudTrail logs 319
Lookout for Vision information in CloudTrail 320
Understanding Lookout for Vision log file entries 321
AWS CloudFormation resources 323
Lookout for Vision and AWS CloudFormation templates 323
Learn more about AWS CloudFormation 323
AWS PrivateLink 324
Considerations for Lookout for Vision VPC endpoints 324
Creating an interface VPC endpoint for Lookout for Vision 324
Creating a VPC endpoint policy for Lookout for Vision 325
Quotas 327
Model quotas 327
Document history 329
AWS Glossary 335

End of support notice: On October 31, 2025, AWS will discontinue support for Amazon Lookout for Vision. After October 31, 2025, you will no longer be able to access the Lookout for Vision console or Lookout for Vision resources. For more information, visit this blog post.

What is Amazon Lookout for Vision?

You can use Amazon Lookout for Vision to find visual defects in industrial products, accurately and at scale. It uses computer vision to identify missing components in an industrial product, damage to vehicles or structures, irregularities in production lines, and even minuscule defects in silicon wafers—or any other physical item where quality is important such as a missing capacitor on printed circuit boards.

Key benefits

Amazon Lookout for Vision provides the following benefits:

- **Quickly and efficiently improve processes** – You can use Amazon Lookout for Vision to implement computer vision-based inspection in industrial processes quickly and efficiently, at scale. You can provide as few as 30 baseline good images and Lookout for Vision can automatically build a custom ML model for defect detection. You can then process images from IP cameras, in batch or in real time, to quickly and accurately identify anomalies like dents, cracks, and scratches.
- **Increase production quality, fast** – With Amazon Lookout for Vision you can reduce defects in production processes, in real time. It identifies and reports visual anomalies in a dashboard so you can take action quickly to stop more defects from occurring—increasing production quality and reducing costs.
- **Reduce operational costs** – Amazon Lookout for Vision reports trends in your visual inspection data, such as identifying processes with the highest defect rate or flagging recent variations in defects. Using this information, you can determine whether to schedule maintenance on the process line or reroute production to another machine before costly, unplanned downtime occurs.

Are you a first-time Amazon Lookout for Vision end user?

If you're a first-time user of Amazon Lookout for Vision, we recommend that you read the following sections in order:

1. **Setting up Amazon Lookout for Vision** – In this section, you set your account details.
2. **Getting started with Amazon Lookout for Vision** – In this section, you learn about creating your first Amazon Lookout for Vision model.

Setting up Amazon Lookout for Vision

In this section, you sign up for an AWS account and set up Amazon Lookout for Vision.

For information about the AWS Regions that support Amazon Lookout for Vision, see Amazon Lookout for Vision Endpoints and Quotas.

Topics

- Step 1: Create an AWS account
- Step 2: Set up permissions
- Step 3: Create the console bucket
- Step 4: Set up the AWS CLI and AWS SDKs
- Step 5: (Optional) Using your own AWS Key Management Service key

Step 1: Create an AWS account

In this step, you sign up for an AWS account and create an administrative user.

Topics

- Sign up for an AWS account
- Create a user with administrative access

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call or text message and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign

administrative access to a user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing **My Account**.

Create a user with administrative access

After you sign up for an AWS account, secure your AWS account root user, enable AWS IAM Identity Center, and create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the *AWS Sign-In User Guide*.

2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the *IAM User Guide*.

Create a user with administrative access

1. Enable IAM Identity Center.

   For instructions, see Enabling AWS IAM Identity Center in the *AWS IAM Identity Center User Guide*.

2. In IAM Identity Center, grant administrative access to a user.

   For a tutorial about using the IAM Identity Center directory as your identity source, see Configure user access with the default IAM Identity Center directory in the *AWS IAM Identity Center User Guide*.

Sign in as the user with administrative access

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

  For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the *AWS Sign-In User Guide*.

Assign access to additional users

1. In IAM Identity Center, create a permission set that follows the best practice of applying least-privilege permissions.

   For instructions, see Create a permission set in the *AWS IAM Identity Center User Guide*.
2. Assign users to a group, and then assign single sign-on access to the group.

   For instructions, see Add groups in the *AWS IAM Identity Center User Guide*.

Step 2: Set up permissions

To use Amazon Lookout for Vision, you needs access permissions to the Lookout for Vision console, AWS SDK operations, and the Amazon S3 bucket that you use for model training.

> **Note**
>
> If you only use AWS SDK operations, you can use policies that are scoped to AWS SDK operations. For more information, see Set up SDK permissions.

Topics

- Setting console access with AWS managed policies
- Setting Amazon S3 bucket permissions
- Assigning permissions

Setting console access with AWS managed policies

Use the following AWS managed policies to apply appropriate access permissions for the Amazon Lookout for Vision console and SDK operations.

- AmazonLookoutVisionConsoleFullAccess — allows full access to the Amazon Lookout for Vision console and SDK operations. You need `AmazonLookoutVisionConsoleFullAccess` permissions to create the console bucket. For more information, see Step 3: Create the console bucket.
- AmazonLookoutVisionConsoleReadOnlyAccess— allows read-only access to the Amazon Lookout for Vision console and SDK operations.

To assign permissions, see Assigning permissions.

For information about AWS managed policies, see AWS managed policies.

Setting Amazon S3 bucket permissions

Amazon Lookout for Vision uses an Amazon S3 bucket to store the following files:

- Dataset images — Images that are used to train a model. For more information, see Creating your dataset.
- Amazon SageMaker AI Ground Truth format manifest files. For example, the manifest file output from SageMaker AI GroundTruth job. For more information, see Creating a dataset using an Amazon SageMaker AI Ground Truth manifest file.
- The output from model training.

If you use the console, Lookout for Vision creates an Amazon S3 bucket (console bucket) to manage your projects. The `LookoutVisionConsoleReadOnlyAccess` and `LookoutVisionConsoleFullAccess` managed policies include Amazon S3 access permissions for the console bucket.

You can use the console bucket to store dataset images and SageMaker AI Ground Truth format manifest files. Alternatively, You can use a different Amazon S3 bucket. The bucket must be owned by your AWS account and must be located in the AWS Region in which you are using Lookout for Vision.

To use a different bucket, add the following policy to the desired user or group. Replace `amzn-s3-demo-bucket` with the name of the desired bucket. For information about adding IAM policies, see Creating IAM Policies.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "LookoutVisionS3BucketAccessPermissions",
            "Effect": "Allow",
            "Action": [
                "s3:GetBucketLocation",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::amzn-s3-demo-bucket"
            ]
        },
        {
            "Sid": "LookoutVisionS3ObjectAccessPermissions",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:GetObjectVersion",
                "s3:PutObject"
            ],
            "Resource": [
                "arn:aws:s3:::amzn-s3-demo-bucket/*"
            ]
        }
    ]
}
```

To assign permissions, see Assigning permissions.

Assigning permissions

To provide access, add permissions to your users, groups, or roles:

- Users and groups in AWS IAM Identity Center:

Create a permission set. Follow the instructions in Create a permission set in the *AWS IAM Identity Center User Guide*.

- Users managed in IAM through an identity provider:

  Create a role for identity federation. Follow the instructions in Create a role for a third-party identity provider (federation) in the *IAM User Guide*.

- IAM users:
  - Create a role that your user can assume. Follow the instructions in Create a role for an IAM user in the *IAM User Guide*.
  - (Not recommended) Attach a policy directly to a user or add a user to a user group. Follow the instructions in Adding permissions to a user (console) in the *IAM User Guide*.

Step 3: Create the console bucket

To use the Amazon Lookout for Vision console, you need an Amazon S3 bucket that is known as the console bucket. The console bucket stores the following:

- Images that you upload to a dataset with the console.
- Training results for model training that you start with console.
- Trial detection results.
- Temporary manifest files that the console creates when you use the console to create a dataset by automatically labeling images in an S3 bucket. The console doesn't delete the manifest files.

When you first open the Amazon Lookout for Vision console in a new AWS Region, Lookout for Vision creates the console bucket on your behalf. Note the console bucket name because you might need to use the bucket name in AWS SDK operations or console tasks, such as creating a dataset.

Alternatively, you can create the console bucket by using Amazon S3. Use this approach if Amazon S3 bucket policies don't let the Amazon Lookout for Vision console successfully create the console bucket. For example, a policy that disallows the automatic creation of an Amazon S3 bucket.

> **Note**
>
> If you only use the AWS SDK and not the Lookout for Vision console, you don't need to create the console bucket. You can use a different S3 bucket with a name of your choosing.

The format of the console bucket name is `lookoutvision-<region>-<random value>`. The random value ensures that there isn't a collision between bucket names.

Topics

- Creating the console bucket with the Amazon Lookout for Vision console
- Creating the console bucket with Amazon S3
- Console bucket settings

Creating the console bucket with the Amazon Lookout for Vision console

Use the following procedure to create the console bucket for an AWS Region with the Amazon Lookout for Vision console. For information about the S3 bucket settings that we enable, see Console bucket settings.

To create the console bucket by using the Amazon Lookout for Vision console

1. Ensure that the user or group you are using has `AmazonLookoutVisionConsoleFullAccess` permission. For more information, see Step 2: Set up permissions.
2. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
3. On the navigation bar, choose **Select a region**. Then choose the AWS Region for which you want to create the console bucket.
4. Choose **Get started**.
5. If this is the first time that you've opened the console in the current AWS Region, do the following in the **First time set up** dialog box:

   a. Copy down the name of the Amazon S3 bucket that's shown. You'll need this information later.

   b. Choose **Create S3 bucket** to let Amazon Lookout for Vision create the console bucket on your behalf.

   The **First time set up** dialog box is not shown if the console bucket for the current AWS Region already exists.

6. Close the browser window.

Creating the console bucket with Amazon S3

You can use Amazon S3 to create the console bucket. You must create the bucket with Amazon S3 versioning enabled. We recommend that you use an Amazon S3 lifecycle configuration to remove noncurrent (previous) versions of an object and delete incomplete multipart uploads. We don't recommend a lifecycle configuration that deletes current versions of an object. For information about the S3 bucket settings we enable for console buckets that you create with the Amazon Lookout for Vision console, see Console bucket settings.

1. Decide the AWS Region in which you want to create a console bucket. For information about supported regions, see Amazon Lookout for Vision endpoints and quotas.
2. Create a bucket using the S3 console instructions at Creating a bucket. Do the following:
   a. For step 3 specify a bucket name that is prepended with `lookoutvision-`*`region`*`-`*`your-identifier`*. Change `region` to the region code you chose in the previous step. Change `your-identifier` to a unique identifier of your choosing.
   b. For step 4 choose the AWS Region that you want to use.
3. Enable versioning for the bucket by following the S3 console instructions at Enabling versioning on buckets.
4. (Optional) Specify a lifecycle configuration for the bucket by following the S3 console instructions at Setting lifecycle configuration on a bucket. Do the following to remove noncurrent (previous) versions of an object and delete incomplete multipart uploads. You don't need to do steps 6, 8, 9, 10.
   a. For step 5 choose **Apply to all objects in the bucket**.
   b. For step 7 select **Permanently delete noncurrent versions of objects** and **Delete expired object delete markers or incomplete multipart uploads**.
   c. For step 11 enter the numbers of days to wait before deleting noncurrent versions of an object.
   d. For step 12 enter the number of days to wait before deleting incomplete multipart uploads.

Console bucket settings

If you create the console bucket with the Amazon Lookout for Vision console, we enable the following settings on the console bucket.

- Versioning of objects in the console bucket.
- Server-side encryption of objects in the console bucket.
- A lifecycle configuration for the deletion of noncurrent objects (30 days) and incomplete multipart uploads (3 days).
- Block public access to the console bucket.

Step 4: Set up the AWS CLI and AWS SDKs

The following steps show you how to install the AWS Command Line Interface (AWS CLI) and AWS SDKs. The examples in this documentation use the AWS CLI, Python, and Java AWS SDKs.

Topics

- Install the AWS SDKS
- Grant programmatic access
- Set up SDK permissions
- Call an Amazon Lookout for Vision operation

Install the AWS SDKS

Follow the steps to download and configure the AWS SDKs.

To set up the AWS CLI and the AWS SDKs

- Download and install the AWS CLI and the AWS SDKs that you want to use. This guide provides examples for the AWS CLI, Java, and Python. For information about installing AWS SDKs, see Tools for Amazon Web Services.

Grant programmatic access

You can run the AWS CLI and code examples in this guide on your local computer or other AWS environments, such as an Amazon Elastic Compute Cloud instance. To run the examples, you need to grant access to the AWS SDK operations that the examples use.

Topics

- Running code on your local computer
- Running code in AWS environments

Running code on your local computer

To run code on a local computer, we recommend that you use short-term credentials to grant a user access to AWS SDK operations. For specific information about running the AWS CLI and code examples on a local computer, see Using a profile on your local computer.

Users need programmatic access if they want to interact with AWS outside of the AWS Management Console. The way to grant programmatic access depends on the type of user that's accessing AWS.

To grant users programmatic access, choose one of the following options.

Which user needs programmatic access?	To	By
Workforce identity (Users managed in IAM Identity Center)	Use temporary credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions for the interface that you want to use. • For the AWS CLI, see Configuring the AWS CLI to use AWS IAM Identity Center in the *AWS Command Line Interface User Guide*. • For AWS SDKs, tools, and AWS APIs, see IAM Identity Center authentication in

Which user needs programmatic access?	To	By
		the *AWS SDKs and Tools Reference Guide*.
IAM	Use temporary credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions in Using temporary credentials with AWS resources in the *IAM User Guide*.
IAM	(Not recommended) Use long-term credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions for the interface that you want to use. • For the AWS CLI, see Authenticating using IAM user credentials in the *AWS Command Line Interface User Guide*. • For AWS SDKs and tools, see Authenticate using long-term credentials in the *AWS SDKs and Tools Reference Guide*. • For AWS APIs, see Managing access keys for IAM users in the *IAM User Guide*.

Using a profile on your local computer

You can run the AWS CLI and code examples in this guide with the short-term credentials you create in Running code on your local computer. To get the credentials and other settings information, the examples use a profile named `lookoutvision-access` For example:

```
session = boto3.Session(profile_name='lookoutvision-access')
```

```
lookoutvision_client = session.client("lookoutvision")
```

The user that the profile represents must have permissions to call the Lookout for Vision SDK operations and other AWS SDK operations needed by the examples. For more information, see Set up SDK permissions. To assign permissions, see Assigning permissions.

To create a profile that works with the AWS CLI and code examples, choose one of the following. Make sure the name of the profile you create is `lookoutvision-access`.

- Users managed by IAM — Follow the instructions at Switching to an IAM role (AWS CLI).
- Workforce identity (Users managed by AWS IAM Identity Center) — Follow the instructions at Configuring the AWS CLI to use AWS IAM Identity Center. For the code examples, we recommend using an Integrated Development Environment (IDE), which supports the AWS Toolkit enabling authentication through IAM Identity Center. For the Java examples, see Start building with Java. For the Python examples, see Start building with Python. For more information, see IAM Identity Center credentials.

> **Note**
>
> You can use code to get short-term credentials. For more information, see Switching to an IAM role (AWS API). For IAM Identity Center, get the short-term credentials for a role by following the instructions at Getting IAM role credentials for CLI access.

Running code in AWS environments

You shouldn't use user credentials to sign AWS SDK calls in AWS environments, such as production code running in an AWS Lambda function. Instead, you configure a role that defines the permissions that your code needs. You then attach the role to the environment that your code runs in. How you attach the role and make temporary credentials available varies depending on the environment that your code runs in:

- AWS Lambda function — Use the temporary credentials that Lambda automatically provides to your function when it assumes the Lambda function's execution role. The credentials are available in the Lambda environment variables. You don't need to specify a profile. For more information, see Lambda execution role.

- Amazon EC2 — Use the Amazon EC2 instance metadata endpoint credentials provider. The provider automatically generates and refreshes credentials for you using the Amazon EC2 *instance profile* you attach to the Amazon EC2 instance. For more information, see Using an IAM role to grant permissions to applications running on Amazon EC2 instances
- Amazon Elastic Container Service — Use the Container credentials provider. Amazon ECS sends and refreshes credentials to a metadata endpoint. A *task IAM role* that you specify provides a strategy for managing the credentials that your application uses. For more information, see Interact with AWS services.
- Greengrass core device — Use X.509 certificates to connect to AWS IoT Core using TLS mutual authentication protocols. These certificates let devices interact with AWS IoT without AWS credentials. The AWS IoT credentials provider authenticates devices using the X.509 certificate and issues AWS credentials in the form of a temporary, limited-privilege security token. For more information, see Interact with AWS services.

For more information about credential providers, see Standardized credential providers.

Set up SDK permissions

To use Amazon Lookout for Vision SDK operations, you need access permissions to the Lookout for Vision API and the Amazon S3 bucket used for model training.

Topics

- Granting SDK operation permissions
- Granting Amazon S3 Bucket permissions
- Assigning permissions

Granting SDK operation permissions

We recommend that you grant only the permissions required to perform a task (least-privilege permissions). For example, to call DetectAnomalies, you need permission to perform `lookoutvision:DetectAnomalies`. To find the permissions for an operation, check the API reference.

When you are just starting out with an application, you might not know the specific permissions you need, so you can start with broader permissions. AWS managed policies provide permissions to help you get started.

- AmazonLookoutVisionFullAccess — allows full access to Amazon Lookout for Vision SDK operations.
- AmazonLookoutVisionReadOnlyAccess — allows access to the read-only SDK operations.

The managed policies for the console also provide access permissions for SDK operations. For more information, see Step 2: Set up permissions.

For information about AWS managed policies, see AWS managed policies.

When you know the permissions that your application needs, reduce permissions further by defining customer managed policies specific to your use cases. For more information, see Customer managed policies.

> **Note**
>
> The getting started instructions require `s3:PutObject` permissions. For more information, see Step 1: Create the manifest file and upload images.

To assign permissions, see Assigning permissions.

Granting Amazon S3 Bucket permissions

To train a model, you need an Amazon S3 bucket with appropriate permissions to store the images, manifest files and training output. The bucket must be owned by your AWS account and must be located in the AWS Region in which you are using Amazon Lookout for Vision.

The SDK-only managed policies (`AmazonLookoutVisionFullAccess` and `AmazonLookoutVisionReadOnlyAccess`) don't include Amazon S3 bucket permissions and you need to apply the following permission policy to access the buckets you use, including existing console buckets.

The console managed policies (`AmazonLookoutVisionConsoleFullAccess` and `AmazonLookoutVisionConsoleReadOnlyAccess`) include access permissions to the console bucket. If you are accessing the console bucket with SDK operations and have console managed policy permissions, you don't need to use the following policy. For more information, see Step 2: Set up permissions.

Deciding task permissions

Use the following information to decide which permissions are needed for the tasks you want to do.

Creating a dataset

To create a dataset with CreateDataset, you need the following permissions.

- `s3:GetBucketLocation` — allows Lookout for Vision to validate that your bucket is in the same region in which you are using Lookout for Vision.
- `s3:GetObject` — Allows access to the manifest file specifed in the `DatasetSource` input parameter. If you want to specify an exact S3 object version of the manifest file, you also need `s3:GetObjectVersion` on the manifest file. For more information, see Using versioning in S3 buckets.

Creating a model

To create a model with CreateModel, you need the following permissions.

- `s3:GetBucketLocation` — allows Lookout for Vision to validate that your bucket is in the same region in which you are using Lookout for Vision.
- `s3:GetObject` — allows access to the images specified in the project's training and test datasets.
- `s3:PutObject` — allows permission to store training output in the specified bucket. You specify the output bucket location in the `OutputConfig` parameter. Optionally, you can scope permissions down to only object keys specified in the `Prefix` field of the `S3Location` input field. For more information, see OutputConfig.

Accessing images, manifest files, and training output

Amazon S3 bucket permissions aren't required to view Amazon Lookout for Vision operation responses. You do need `s3:GetObject` permission if you want to access images, manifests files, and training output referenced in operation responses. If you are accessing a versioned Amazon S3 object, you need `s3:GetObjectVersion` permission.

Setting Amazon S3 bucket policy

You can use the following policy to specify the Amazon S3 bucket permissions needed to create a dataset (`CreateDataset`), create a model (`CreateModel`), and access images, manifest files, and training output. Change the value of *`amzn-s3-demo-bucket`* to the name of the bucket that you want use.

You can adjust the policy to your needs. For more information, see Deciding task permissions. Add the policy to the desired user. For more information, see Creating IAM Policies.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "LookoutVisionS3BucketAccess",
            "Effect": "Allow",
            "Action": "s3:GetBucketLocation",
            "Resource": [
                "arn:aws:s3:::amzn-s3-demo-bucket"
            ],
            "Condition": {
                "Bool": {
                    "aws:ViaAWSService": "true"
                }
            }
        },
        {
            "Sid": "LookoutVisionS3ObjectAccess",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:GetObjectVersion",
                "s3:PutObject"
            ],
            "Resource": [
                "arn:aws:s3:::amzn-s3-demo-bucket/*"
            ],
            "Condition": {
                "Bool": {
                    "aws:ViaAWSService": "true"
                }
            }
        }
```

```
    ]
}
```

To assign permissions, see Assigning permissions.

Assigning permissions

To provide access, add permissions to your users, groups, or roles:

- Users and groups in AWS IAM Identity Center:

  Create a permission set. Follow the instructions in Create a permission set in the *AWS IAM Identity Center User Guide*.
- Users managed in IAM through an identity provider:

  Create a role for identity federation. Follow the instructions in Create a role for a third-party identity provider (federation) in the *IAM User Guide*.
- IAM users:
  - Create a role that your user can assume. Follow the instructions in Create a role for an IAM user in the *IAM User Guide*.
  - (Not recommended) Attach a policy directly to a user or add a user to a user group. Follow the instructions in Adding permissions to a user (console) in the *IAM User Guide*.

Call an Amazon Lookout for Vision operation

Run the following code to confirm that you can make calls to the Amazon Lookout for Vision API. The code lists the projects in your AWS account, in the current AWS Region. If you haven't previously created a project, the response is empty, but does confirm that you can call the `ListProjects` operation.

In general, calling an example function requires an AWS SDK Lookout for Vision client and any other required parameters. The AWS SDK Lookout for Vision client is declared in the main function.

If the code fails, check that the user that you use has the correct permissions. Also check the AWS Region that you using as Amazon Lookout for Vision is not available in all AWS Regions.

To call an Amazon Lookout for Vision operation

1. If you haven't already done so, install and configure the AWS CLI and the AWS SDKs. For more information, see Step 4: Set up the AWS CLI and AWS SDKs.
2. Use the following example code to view your projects.

   CLI

   Use the `list-projects` command to list the projects in your account.

```
aws lookoutvision list-projects \
--profile lookoutvision-access
```

   Python

   This code is taken from the AWS Documentation SDK examples GitHub repository. See the full example here.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0


from botocore.exceptions import ClientError
import boto3

class GettingStarted:

    @staticmethod
    def list_projects(lookoutvision_client):
        """
        Lists information about the projects that are in in your AWS account
        and in the current AWS Region.

        :param lookoutvision_client: A Boto3 Lookout for Vision client.
        """
        try:
            response = lookoutvision_client.list_projects()
            for project in response["Projects"]:
                print("Project: " + project["ProjectName"])
                print("ARN: " + project["ProjectArn"])
```

```
                print()
            print("Done!")
        except ClientError:
            raise
def main():
    session = boto3.Session(profile_name='lookoutvision-access')
    lookoutvision_client = session.client("lookoutvision")

    GettingStarted.list_projects(lookoutvision_client)

if __name__ == "__main__":
    main()
```

Java V2

This code is taken from the AWS Documentation SDK examples GitHub repository. See the full example here.

```
/*
   Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
   SPDX-License-Identifier: Apache-2.0
*/

package com.example.lookoutvision;

import software.amazon.awssdk.auth.credentials.ProfileCredentialsProvider;
import software.amazon.awssdk.services.lookoutvision.LookoutVisionClient;
import software.amazon.awssdk.services.lookoutvision.model.ProjectMetadata;
import
 software.amazon.awssdk.services.lookoutvision.paginators.ListProjectsIterable;
import software.amazon.awssdk.services.lookoutvision.model.ListProjectsRequest;
import
 software.amazon.awssdk.services.lookoutvision.model.LookoutVisionException;

import java.util.ArrayList;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;

public class GettingStarted {
```

```
   public static final Logger logger =
Logger.getLogger(GettingStarted.class.getName());

   /**
    * Lists the Amazon Lookoutfor Vision projects in the current AWS account
and
    * AWS Region.
    *
    * @param lfvClient   An Amazon Lookout for Vision client.
    * @return List<ProjectMetadata> Metadata for each project.
    */
   public static List<ProjectMetadata> listProjects(LookoutVisionClient
lfvClient)
           throws LookoutVisionException {

       logger.log(Level.INFO, "Getting projects:");
       ListProjectsRequest listProjectsRequest = ListProjectsRequest.builder()
               .maxResults(100)
               .build();

       List<ProjectMetadata> projectMetadata = new ArrayList<>();

       ListProjectsIterable projects =
lfvClient.listProjectsPaginator(listProjectsRequest);

       projects.stream().flatMap(r -> r.projects().stream())
               .forEach(project -> {
                   projectMetadata.add(project);
                   logger.log(Level.INFO, project.projectName());
               });

       logger.log(Level.INFO, "Finished getting projects.");

       return projectMetadata;

   }

   public static void main(String[] args) throws Exception {

       try {

           // Get the Lookout for Vision client.
           LookoutVisionClient lfvClient = LookoutVisionClient.builder()
```

```
 .credentialsProvider(ProfileCredentialsProvider.create("lookoutvision-access"))
                    .build();

            List<ProjectMetadata> projects = Projects.listProjects(lfvClient);

            System.out.printf("Projects%n--------%n");

            for (ProjectMetadata project : projects) {
                System.out.printf("Name: %s%n", project.projectName());
                System.out.printf("ARN: %s%n", project.projectArn());
                System.out.printf("Date: %s%n%n",
 project.creationTimestamp().toString());
            }

        } catch (LookoutVisionException lfvError) {
            logger.log(Level.SEVERE, "Could not list projects: {0}: {1}",
                    new Object[] { lfvError.awsErrorDetails().errorCode(),
                            lfvError.awsErrorDetails().errorMessage() });
            System.out.println(String.format("Could not list projects: %s",
 lfvError.getMessage()));
            System.exit(1);
        }

    }

}
```

Step 5: (Optional) Using your own AWS Key Management Service key

You can use AWS Key Management Service (KMS) to manage encryption for the input images that you store in Amazon S3 buckets.

By default your images are encrypted with a key that AWS owns and manages. You can also choose to use your own AWS Key Management Service (KMS) key. For more information, see AWS Key Management Service concepts.

If you want to use your own KMS key, use the following policy to specify the KMS key. Change *kms_key_arn* to the ARN of the KMS key (or KMS alias ARN) that you want to use. Alternatively,

specify * to use any KMS key. For information about adding the policy to a user or role, see Creating IAM Policies.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "LookoutVisionKmsDescribeAccess",
            "Effect": "Allow",
            "Action": "kms:DescribeKey",
            "Resource": "kms_key_arn"
        },
        {
            "Sid": "LookoutVisionKmsCreateGrantAccess",
            "Effect": "Allow",
            "Action": "kms:CreateGrant",
            "Resource": "kms_key_arn",
            "Condition": {
                "StringLike": {
                    "kms:ViaService": "lookoutvision.*.amazonaws.com"
                },
                "Bool": {
                    "kms:GrantIsForAWSResource": "true"
                }
            }
        }
    ]
}
```

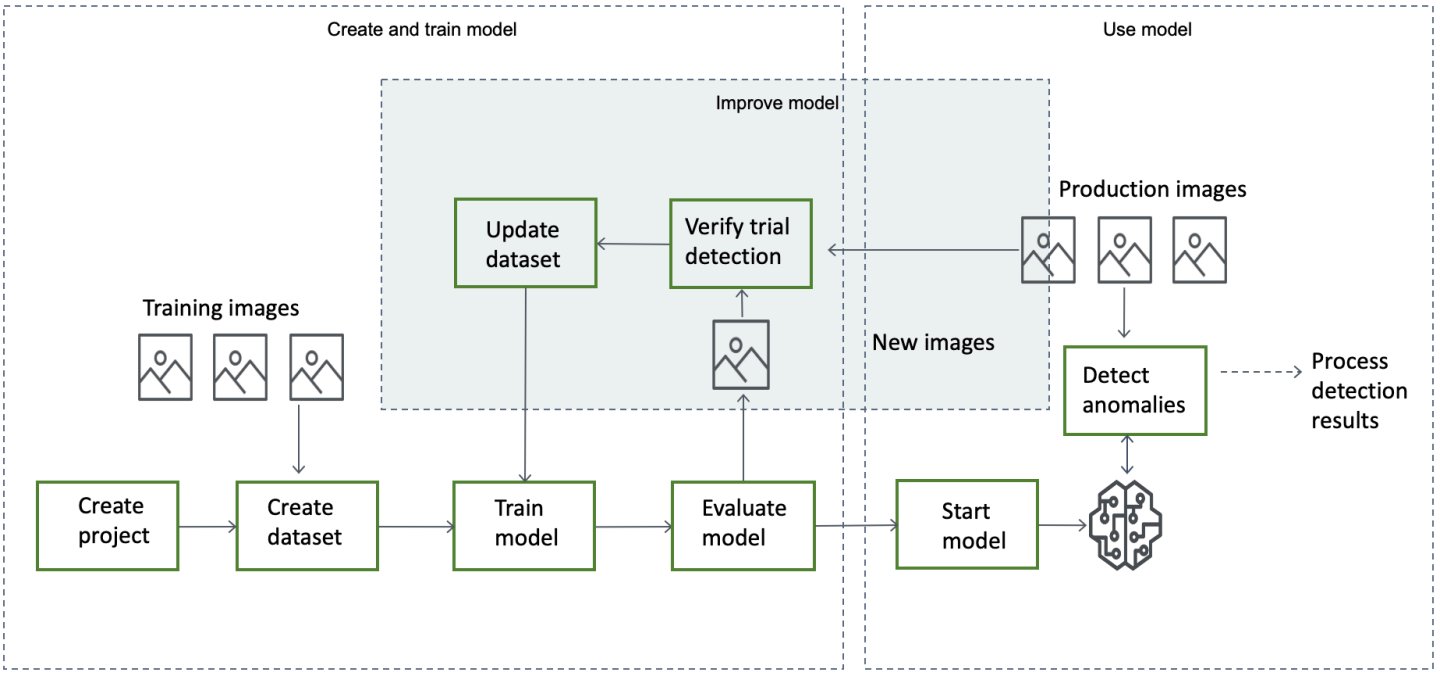
Understanding Amazon Lookout for Vision

You can use Amazon Lookout for Vision to find visual defects in industrial products, accurately and at scale, for tasks such as:

- **Detecting damaged parts** – Spot damage to a product's surface quality, color, and shape during the fabrication and assembly process.
- **Identifying missing components** – Determine missing components based on the absence, presence, or placements of objects. For example, a missing capacitor on a printed circuit board.
- **Uncovering process issues** – Detect defects with repeating patterns, such as repeated scratches in the same spot on a silicon wafer.

With Lookout for Vision you create a computer vision model that predicts the presence of anomalies in an image. You provide the images that Amazon Lookout for Vision uses to train and test your model. Amazon Lookout for Vision provides metrics that you can use to evaluate and improve your trained model. You can host the trained model in the AWS cloud or you can deploy the model to an edge device. A simple API operation returns the predictions that your model makes.

The general workflow for creating, evaluating, and using a model is as follows:



Topics

- Choose your model type
- Create your model
- Evaluate your model
- Use your model
- Use your model on an edge device
- Use your dashboard

Choose your model type

Before you can create a model, you must decide which type of model you want. You can create two types of model, *image classification* and *image segmentation*. You decide which type of model to create based on your use case.

Image classification model

If you only need to know if an image contains an anomaly, but don't need to know its location, create an image classification model. An image classification model makes a prediction of whether an image contains an anomaly. The prediction includes the model's confidence in the accuracy of the prediction. The model doesn't provide any information about the location of any anomalies found on the image.

Image segmentation model

If you need to know the location of an anomaly, such as the location of a scratch, create an image segmentation model. Amazon Lookout for Vision models use *semantic segmentation* to identify the pixels on an image where the types of anomalies (such as a scratch or a missing part) are present.

> **Note**
>
> A semantic segmentation model locates different types of anomaly. It doesn't provide instance information for individual anomalies. For example, if an image contains two *dents*, Lookout for Vision returns information about both dents in a single entity representing the *dent* anomaly type.

An Amazon Lookout for Vision segmentation model predicts the following:

Classification

The model returns a classification for an analyzed image (normal/anomaly), which includes the model's confidence in the prediction. Classification information is calculated separately from segmentation information and you shouldn't assume a relationship between them.

Segmentation

The model returns an image mask that marks the pixels where anomalies occur on the image. Different types of anomaly are color coded according to the color assigned to the *anomaly label* in the dataset. An anomaly label represents the type of an anomaly. For example, the blue mask in the following image marks the location of a *scratch* anomaly type found on a car.

The model returns the color code for each anomaly label in the mask. The model also returns the percentage covering of the image that an anomaly label has.

With a Lookout for Vision segmentation model, you can use various criteria to analyze the analysis results from the model. For example:

- Anomaly location – If you need to know the location of anomalies, use segmentation information to see masks that cover anomalies.
- Types of anomaly – Use segmentation information to decide if an image contains more than an acceptable number of anomaly types.
- Area of coverage – Use segmentation information to decide if an anomaly type covers more than an acceptable area of an image.
- Image classification – If you don't need to know the location of anomalies, use classification information to determine if an image contains anomalies.

For example code, see Detecting anomalies in an image.

After you decide which type of model you want, you create a project and a dataset to manage your model. Using Labels, you can classify images as normal or an anomaly. Labels also identify segmentation information such as masks and anomaly types. How you label the images in your dataset determines the type of model that Lookout for Vision creates for you.

Labeling an image segmentation model is more complex than labeling an image classification model. To train a segmentation model, you have to classify the training images as normal or anomalous. You also have to define anomaly masks and anomaly types for each anomalous image. A classification model only requires you to identify training images as normal or anomalous.

Create your model

The steps to create a model are creating a project, creating a dataset, and training the model are as follows:

Create a project

Create a project to manage the datasets and the models that you create. A project should be used for a single use case, such as detecting anomalies in a single type of machine part.

You can use the dashboard to get an overview of your projects. For more information, see Using the Amazon Lookout for Vision dashboard.

More information: Create your project.

Create a dataset

To train a model Amazon Lookout for Vision needs images of normal and anomalous objects for your use case. You supply these images in a *dataset*.

A dataset is a set of images and labels that describe those images. The images should represent a single type of object on which anomalies can occur. For more information, see Preparing images for a dataset.

With Amazon Lookout for Vision you can have a project that uses a single dataset, or a project that has separate training and test datasets. We recommend using a single dataset project unless you want finer control over training, testing, and performance tuning.

You create a dataset by importing the images. Depending on how you import the images, the images might be also be labeled. If not, you use the console to label the images.

Importing images

If you create the dataset with the Lookout for Vision console, you can import the images in one of the following ways:

- Import images from your local computer. The images aren't labeled.
- Import images from an S3 bucket. Amazon Lookout for Vision can classify the images using the folder names that contain the images. Use `normal` for normal images. Use `anomaly` for anomalous images. You can't automatically assign segmentation labels.
- Import an Amazon SageMaker AI Ground Truth manifest file. Images in a manifest file are labeled. You can create and import your own manifest file. If you have many images, consider using the SageMaker AI Ground Truth labeling service. You then import the output manifest file from the Amazon SageMaker AI Ground Truth job.

Labeling images

Labels describe an image in a dataset. Labels specify if an image is normal or anomalous (classification). Labels also describe the location of anomalies on an image (segmentation).

If your images aren't labeled, you can use the console to label them.

The labels you assign to images in your dataset determines the type of model that Lookout for Vision creates:

Image classification

To create an image classification model, use the Lookout for Vision console to classify images in the dataset as normal or an anomaly.

You can also use the `CreateDataset` operation to create a dataset from a manifest file that includes classification information.

Image segmentation

To create an image segmentation model, use the Lookout for Vision console to classify images in the dataset as normal or an anomaly. You also specify pixel masks for anomalous areas on the image (if they exist) as well as an anomaly label for individual anomaly masks.

You can also use the `CreateDataset` operation to create a dataset from a manifest file that includes segmentation and classification information.

If your project has separate training and test datasets, Lookout for Vision uses the training dataset to learn and determine the model type. You should label the images in your test dataset in the same way.

More information: Creating your dataset.

Train your model

Training creates a model and trains it to predict the presence of anomalies in images. A new version of your model is created each time you train.

At the start of training, Amazon Lookout for Vision chooses the most suitable algorithm to train your model with. The model is trained and then tested. In Getting started with Amazon Lookout for Vision, you train a single dataset project, the dataset is internally split to create a training dataset and a test dataset. You can also create a project that has separate training and test datasets. In this configuration, Amazon Lookout for Vision trains your model with the training dataset and tests the model with the test dataset.

> **Important**
>
> You are charged for the amount of time that it takes to successfully train your model.

More information: Train your model.

Evaluate your model

Evaluate the performance of your model by using the performance metrics created during testing.

Using performance metrics, you can better understand the performance of your trained model, and decide if you're ready to use it in production.

More information: Improving your model.

If the performance metrics indicate that improvements are needed, you can add more training data by running a trial detection task with new images. After the task completes, you can verify the results and add the verified images to your training dataset. Alternatively, you can add new training images directly to the dataset. Next, you retrain your model and recheck the performance metrics.

More information: Verifying your model with a trial detection task .

Use your model

Before you can use your model in the AWS cloud, you start the model with the StartModel operation. You can get the `StartModel` CLI command for your model from the console.

More information: Start your model.

A trained Amazon Lookout for Vision model predicts whether an input image contains normal or anomalous content. If your model is a segmentation model, the prediction includes an anomaly mask that marks the pixels where anomalies are found.

To make a prediction with your model, call the DetectAnomalies operation and pass an input image from your local computer. You can get the CLI command that calls `DetectAnomalies` from the console.

More information: Detect anomalies in an image.

> **Important**
>
> You are charged for the time that your model is running.

If you are no longer using your model, use the StopModel operation to stop the model. You can get the CLI command from the console.

More information: Stop your model.

Use your model on an edge device

You can use a Lookout for Vision model on an AWS IoT Greengrass Version 2 core device.

More information: Using your Amazon Lookout for Vision model on an edge device.

Use your dashboard

You can use the dashboard to get an overview of all your projects and overview information for individual projects.

More information: Use your dashboard.

Getting started with Amazon Lookout for Vision

Before starting these *Getting started* instructions, we recommend that you read Understanding Amazon Lookout for Vision.

The Getting Started instructions show you how to use create an example image segmentation model. If you want to create an example image classification model, see Image classification dataset.

If you want to quickly try an example model, we provide example training images and mask images. We also provide a Python script that creates an image segmentation manifest file. You use the manifest file to create a dataset for your project and you don't need to label the images in the dataset. When you create a model with your own images, you must label the images in the dataset. For more information, see Creating your dataset.

The images we provide are of normal and anomalous cookies. An anomalous cookie has a crack across the cookie shape. The model you train with the images predicts a classification (normal or anomalous) and finds the area (mask) of cracks in an anomalous cookie, as shown in the following example.

Topics

- Step 1: Create the manifest file and upload images
- Step 2: Create the model
- Step 3: Start the model
- Step 4: Analyze an image
- Step 5: Stop the model
- Next steps

Step 1: Create the manifest file and upload images

In this procedure, you clone the Amazon Lookout for Vision documentation repository to your computer. You then use a Python (version 3.7 or higher) script to create a manifest file and upload the training images and mask images to an Amazon S3 location that you specify. You use the manifest file to create your model. Later, you use test images in the local repository to try your model.

To create the manifest file and upload images

1. Set up Amazon Lookout for Vision by following the instructions at Setup Amazon Lookout for Vision. Be sure to install the AWS SDK for Python.
2. In the AWS Region in which you want to use Lookout for Vision, create an S3 bucket.
3. In the Amazon S3 bucket, create a folder named `getting-started`.
4. Note the Amazon S3 URI and Amazon Resource name (ARN) for the folder. You use them to set up permissions and to run the script.
5. Make sure that the user calling the script has permissions to call the `s3:PutObject` operation. You can use the following policy. To assign permissions, see Assigning permissions.

```
{
 "Version": "2012-10-17",
 "Statement": [{
  "Sid": "Statement1",
  "Effect": "Allow",
  "Action": [
   "s3:PutObject"
  ],
  "Resource": [
   "arn:aws:s3::: ARN for S3 folder in step 4/*"
  ]
 }]
}
```

6. Make sure that you have a local profile named `lookoutvision-access` and that the profile user has the permission from the previous step. For more information, see Using a profile on your local computer.
7. Download the zip file, getting-started.zip. The zip file contains the getting started dataset and set up script.

8. Unzip the file `getting-started.zip`.
9. At the command prompt, do the following:

   a. Navigate to the `getting-started` folder.

   b. Run the following command to create a manifest file and upload the training images and image masks to the Amazon S3 path you noted in step 4.

   ```
   python getting_started.py S3-URI-from-step-4
   ```

   c. When the script completes, note the path to the `train.manifest` file that the script displays after `Create dataset using manifest file:`. The path should be similar to `s3://path to getting started folder/manifests/train.manifest`.

Step 2: Create the model

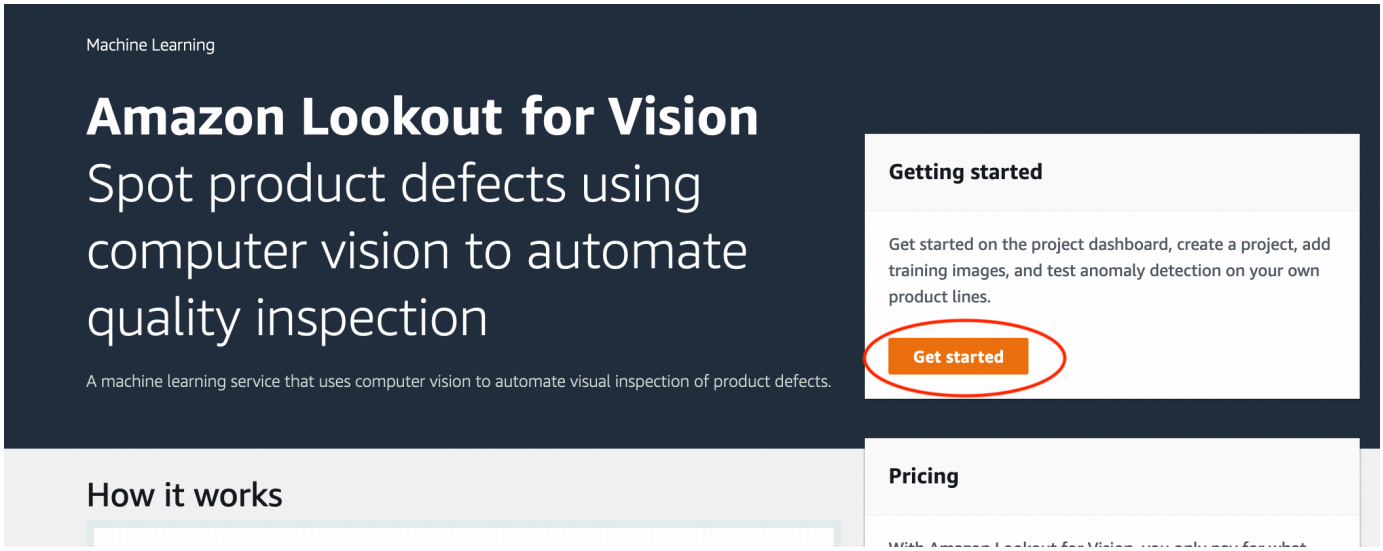
In this procedure, you create a project and dataset using the images and manifest file that you previously uploaded to your Amazon S3 bucket. You then create the model and view the evaluation results from model training.

Because you create the dataset from the getting started manifest file, you don't need to label the dataset's images. When you create a dataset with your own images, you do need to label images. For more information, see Labeling images.

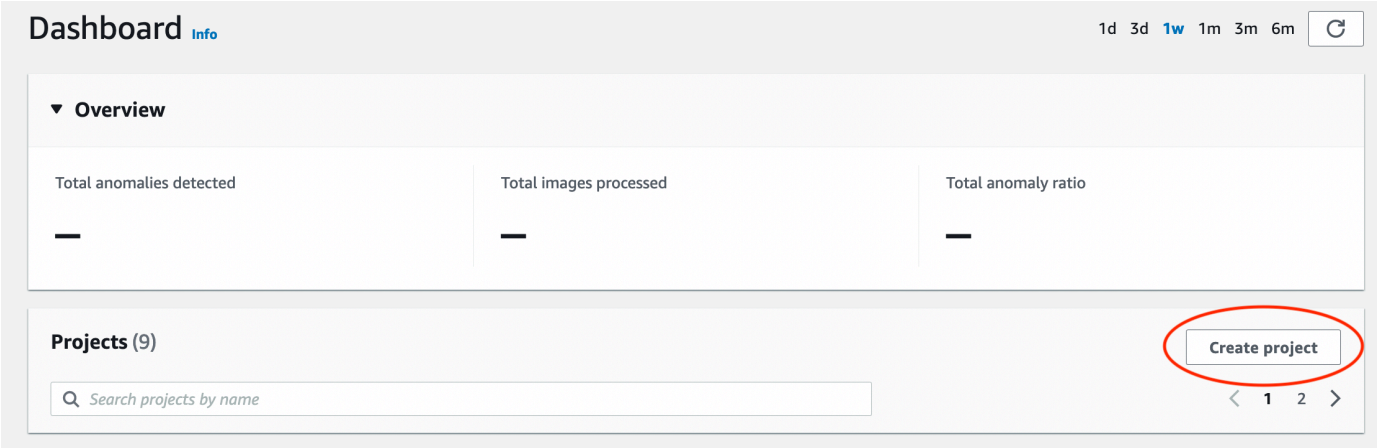
> **Important**
>
> You are charged for a successful training of a model.

To create a model

1. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
2. Make sure you are in the same AWS Region that you created the Amazon S3 bucket in Step 1: Create the manifest file and upload images. To change the Region, choose the name of the currently displayed Region in the navigation bar. Then select the Region to which you want to switch.
3. Choose **Get started**.



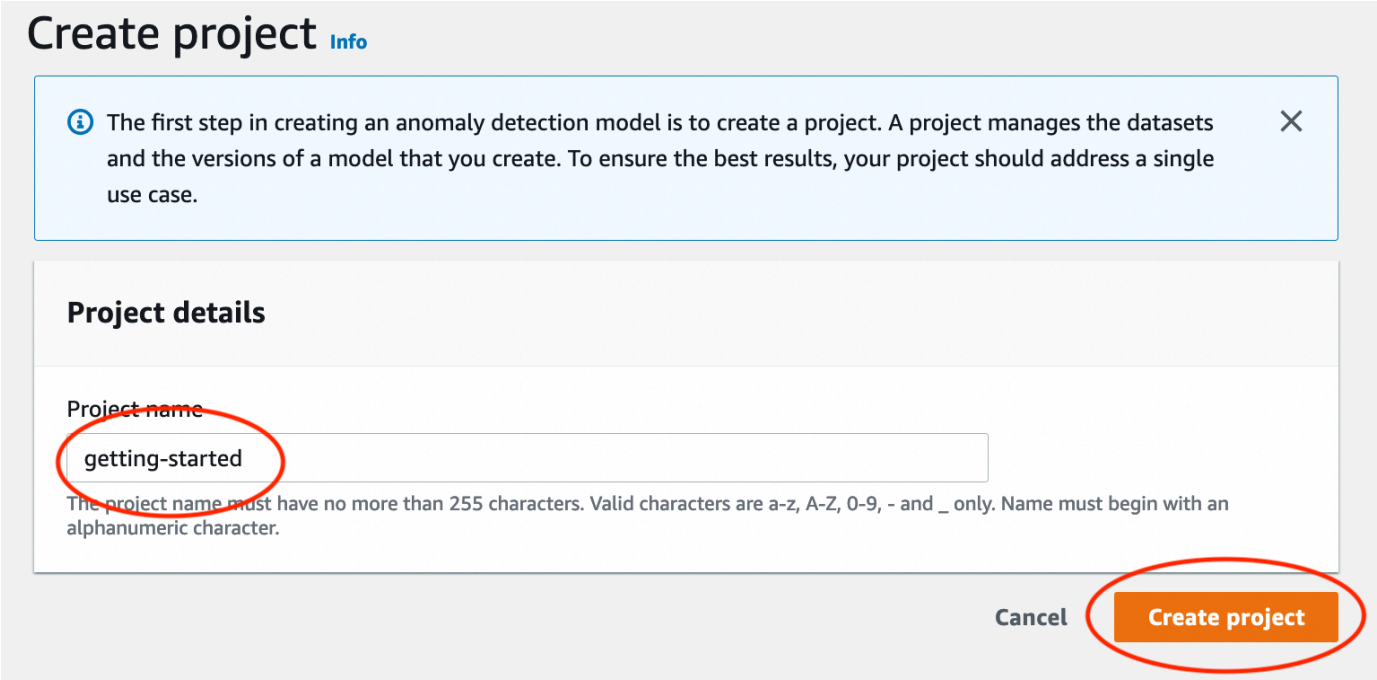
4. In the **Projects** section, choose **Create project**.



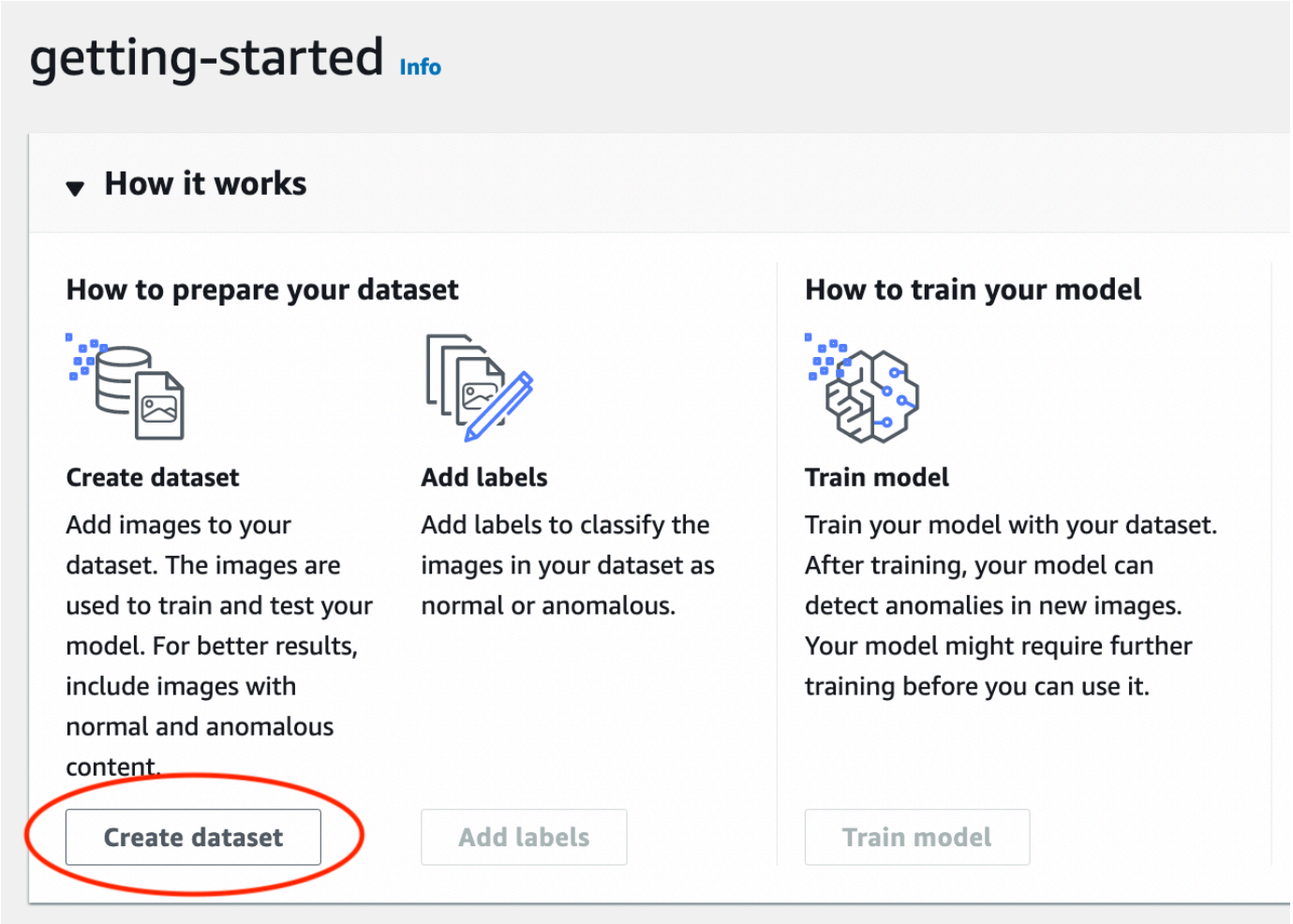
5. On the **Create project** page, do the following:

   a. In **Project name**, enter `getting-started`.

   b. Choose **Create project**.



6. On the project page, in the **How it works** section, choose **Create dataset**.



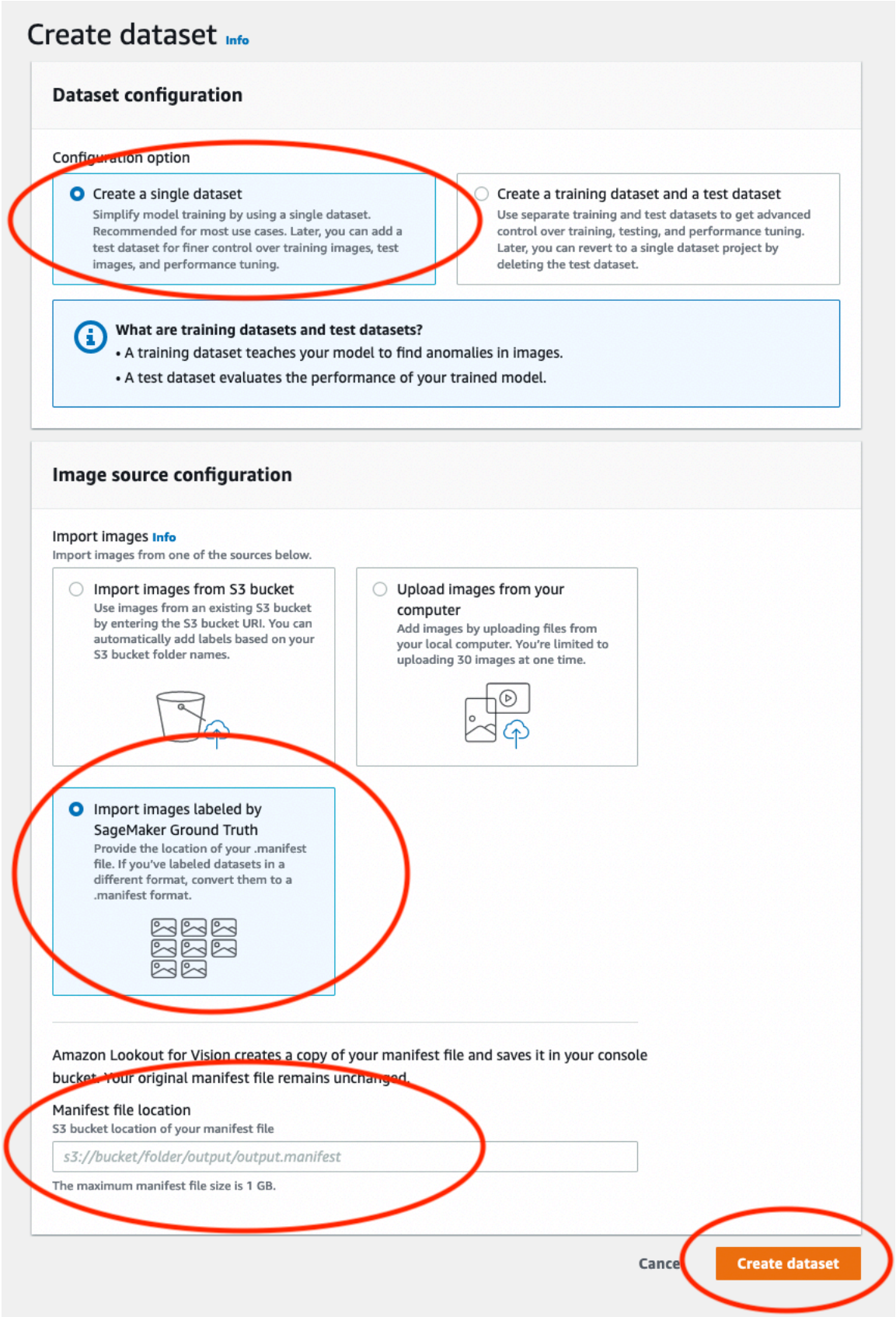
7. On the **Create dataset** page, do the following:

   a. Choose **Create a single dataset**.

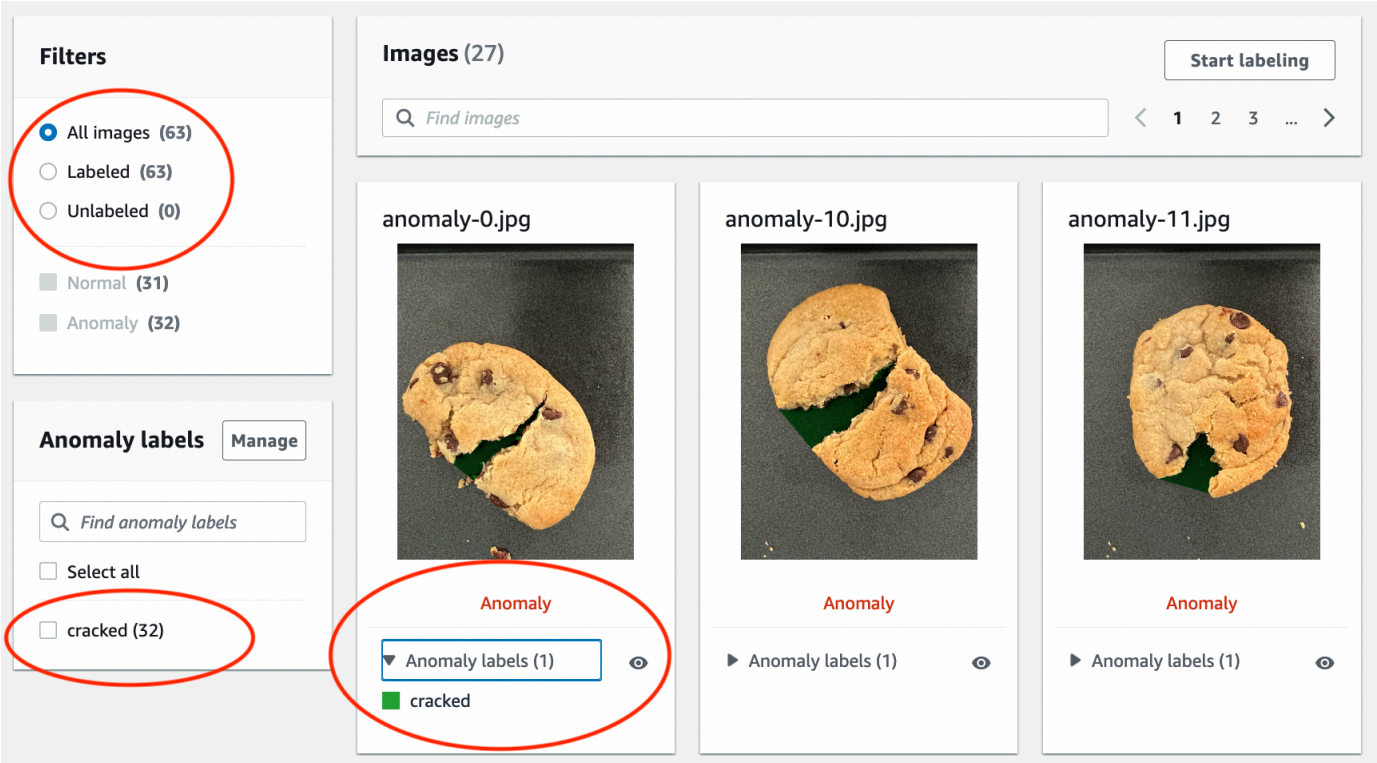
   b. In the **Image source configuration** section, choose **Import images labeled by SageMaker Ground Truth**.

   c. For **.manifest file location**, enter the Amazon S3 location of the manifest file that you noted in step 6.c. of Step 1: Create the manifest file and upload images. The Amazon S3 location should be similar to `s3:`//*path to getting started folder*`/manifests/train.manifest`

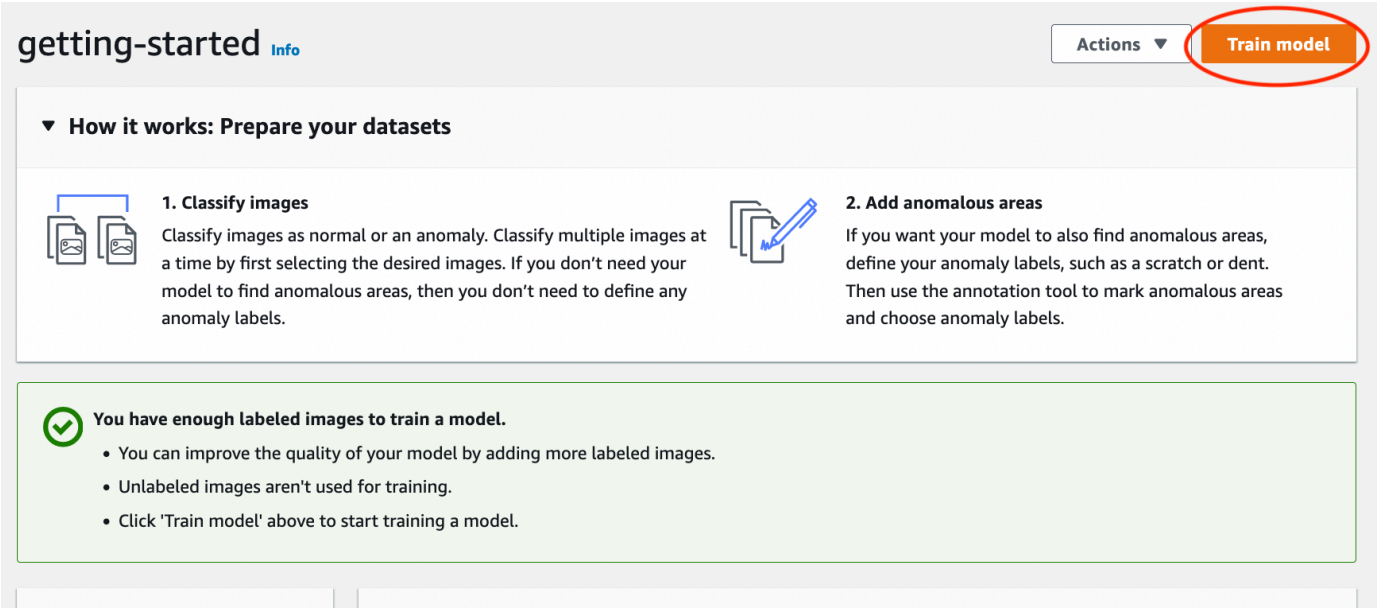
   d. Choose **Create dataset**.


Create dataset Info
Dataset configuration
Configuration option
Create a single dataset
Simplify model training by using a single dataset. Recommended for most use cases. Later, you can add a test dataset for finer control over training images, test images, and performance tuning.
Create a training dataset and a test dataset
Use separate training and test datasets to get advanced control over training, testing, and performance tuning. Later, you can revert to a single dataset project by deleting the test dataset.
What are training datasets and test datasets?
• A training dataset teaches your model to find anomalies in images.
• A test dataset evaluates the performance of your trained model.
Image source configuration
Import images Info
Import images from one of the sources below.
Import images from S3 bucket
Use images from an existing S3 bucket by entering the S3 bucket URI. You can automatically add labels based on your S3 bucket folder names.
Upload images from your computer
Add images by uploading files from your local computer. You're limited to uploading 30 images at one time.
Import images labeled by SageMaker Ground Truth
Provide the location of your .manifest file. If you've labeled datasets in a different format, convert them to a .manifest format.
Amazon Lookout for Vision creates a copy of your manifest file and saves it in your console bucket. Your original manifest file remains unchanged.
Manifest file location
S3 bucket location of your manifest file
s3://bucket/folder/output/output.manifest
The maximum manifest file size is 1 GB.
Cancel
Create dataset

8. On the project details page, in the **Images** section, view the dataset images. You can view the classification and image segmentation information (mask and anomaly labels) for each dataset image. You can also search for images, filter images by labeling status (labeled/unlabeled), or filter images by the anomaly labels assigned to them.

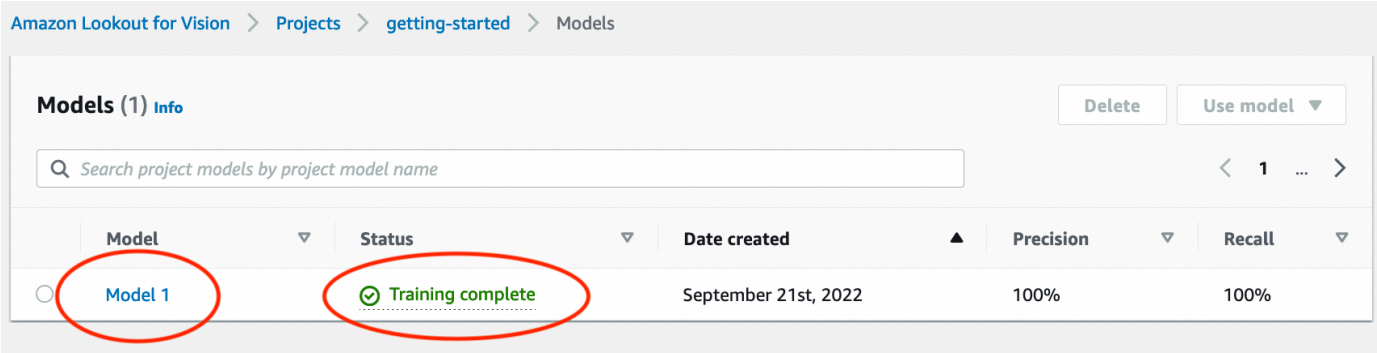


9. On the project details page, choose **Train model**.



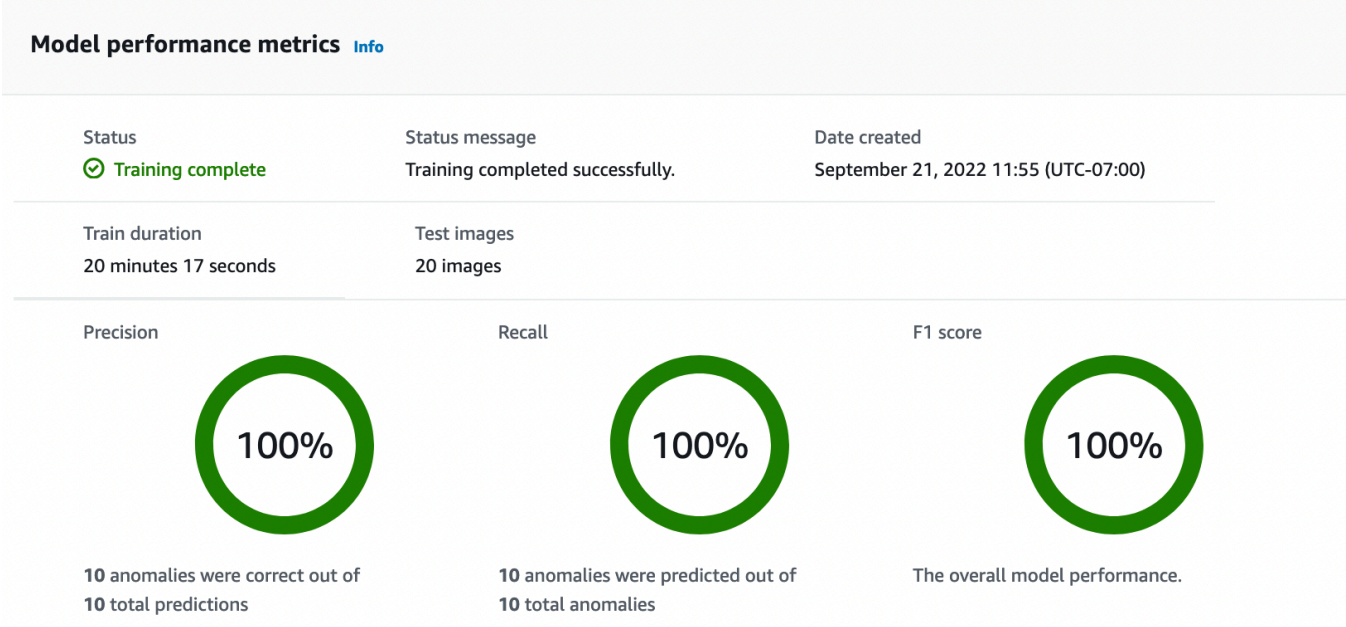
10. On the **Train model** details page, choose **Train model**.

11. In the **Do you want to train your model?** dialog box, choose **Train model**.
12. In the project **Models** page, you can see that training has started. Check the current status by viewing the **Status** column for the model version. Training the model takes at least 30 minutes to complete. Training has successfully finished when the status changes to **Training complete**.
13. When training finishes, choose the model **Model 1** in the **Models** page.

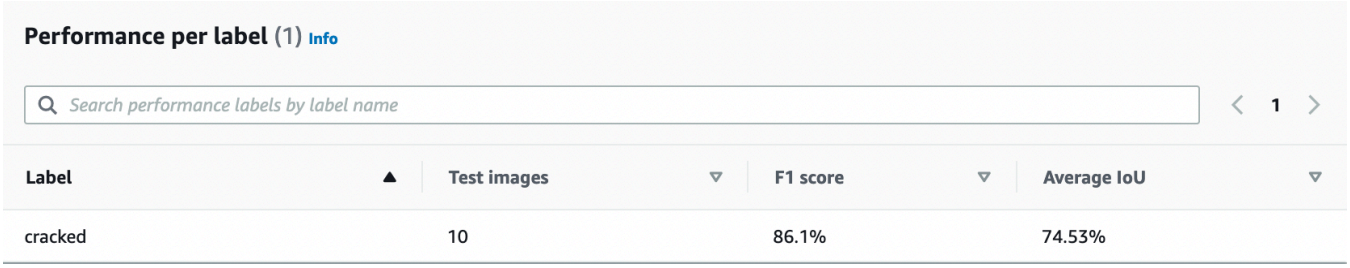


14. In the model's details page, view the evaluation results in the **Performance metrics** tab. There are metrics for the following:

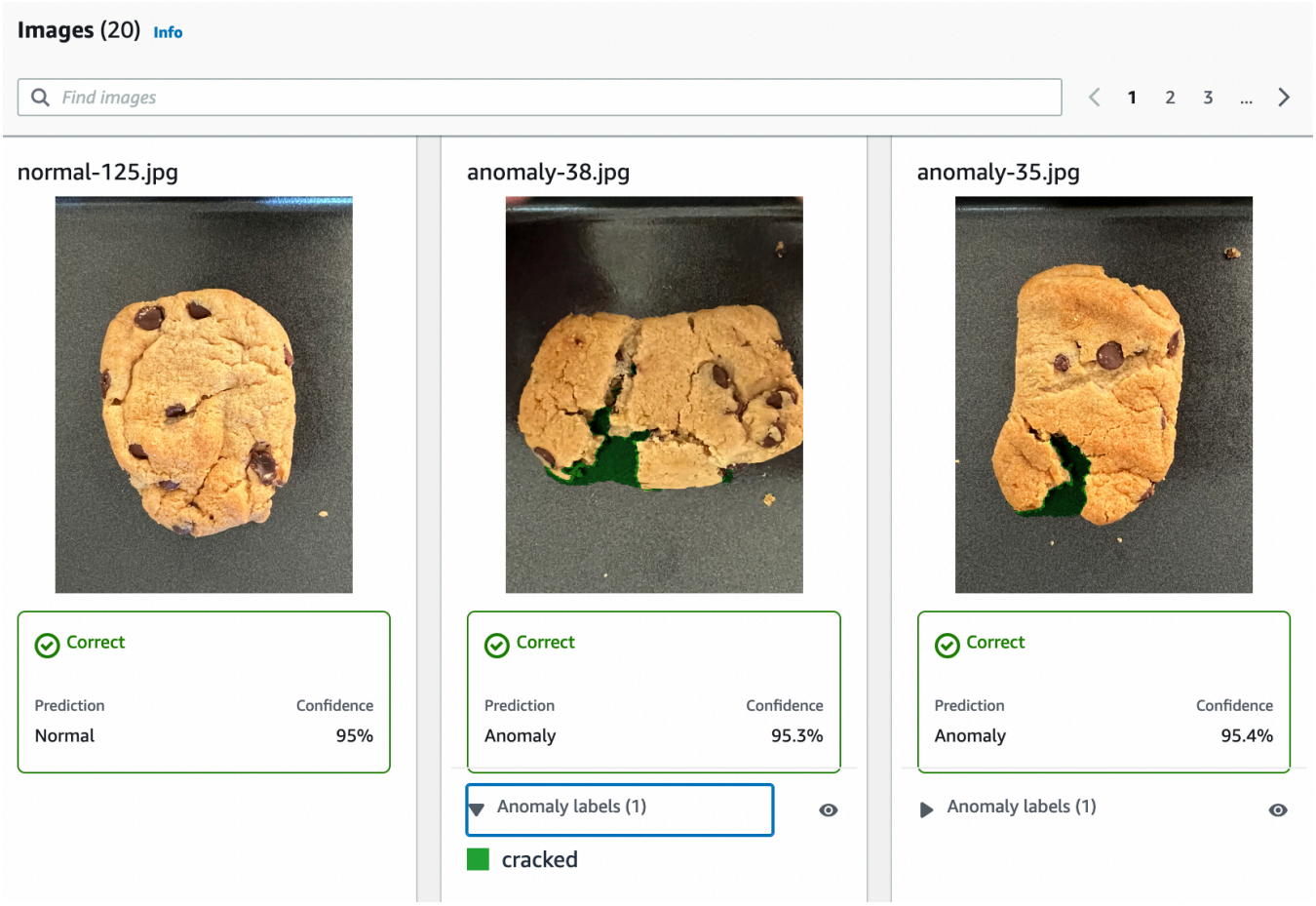
    - Overall model performance metrics (precision, recall, and F1 score) for the classification predictions made by the model.



    - Performance metrics for anomaly labels found in the test images (Average IoU, F1 score)



- Predictions for test images (classification, segmentation masks, and anomaly labels)



As model training is non-deterministic, your evaluation results might differ from the results on shown on this page. For more information, see Improving your Amazon Lookout for Vision model.

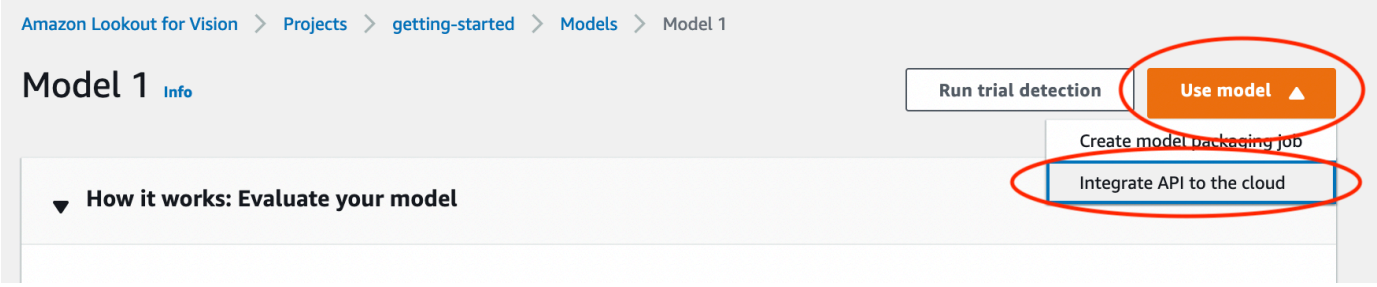
Step 3: Start the model

In this step, you start hosting the model so that it is ready to analyze images. For more information, see Running your trained Amazon Lookout for Vision model.

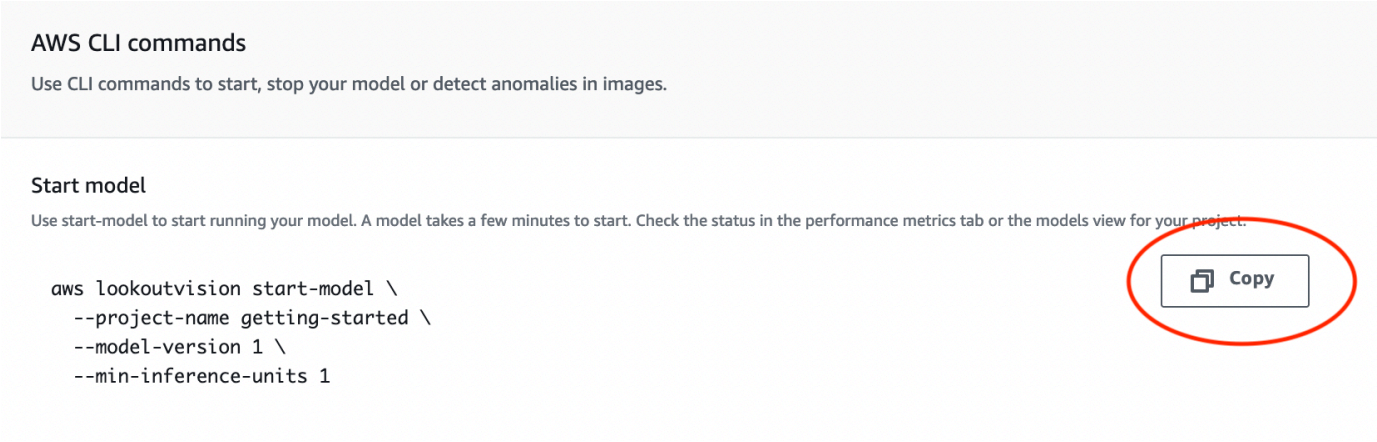
> **Note**
>
> You are charged for the amount of time that your model runs. You stop your model in Step 5: Stop the model.

To start the model.

1. On the model's details page, choose **Use model** and then choose **Integrate API to the cloud**.



2. In the **AWS CLI commands** section, copy the `start-model` AWS CLI command.



3. Make sure that the AWS CLI is configured to run in the same AWS Region in which you are using the Amazon Lookout for Vision console. To change the AWS Region that the AWS CLI uses, see Install the AWS SDKS.
4. At the command prompt, start the model by entering the `start-model` command. If you are using the `lookoutvision` profile to get credentials, add the `--profile lookoutvision-access` parameter. For example:

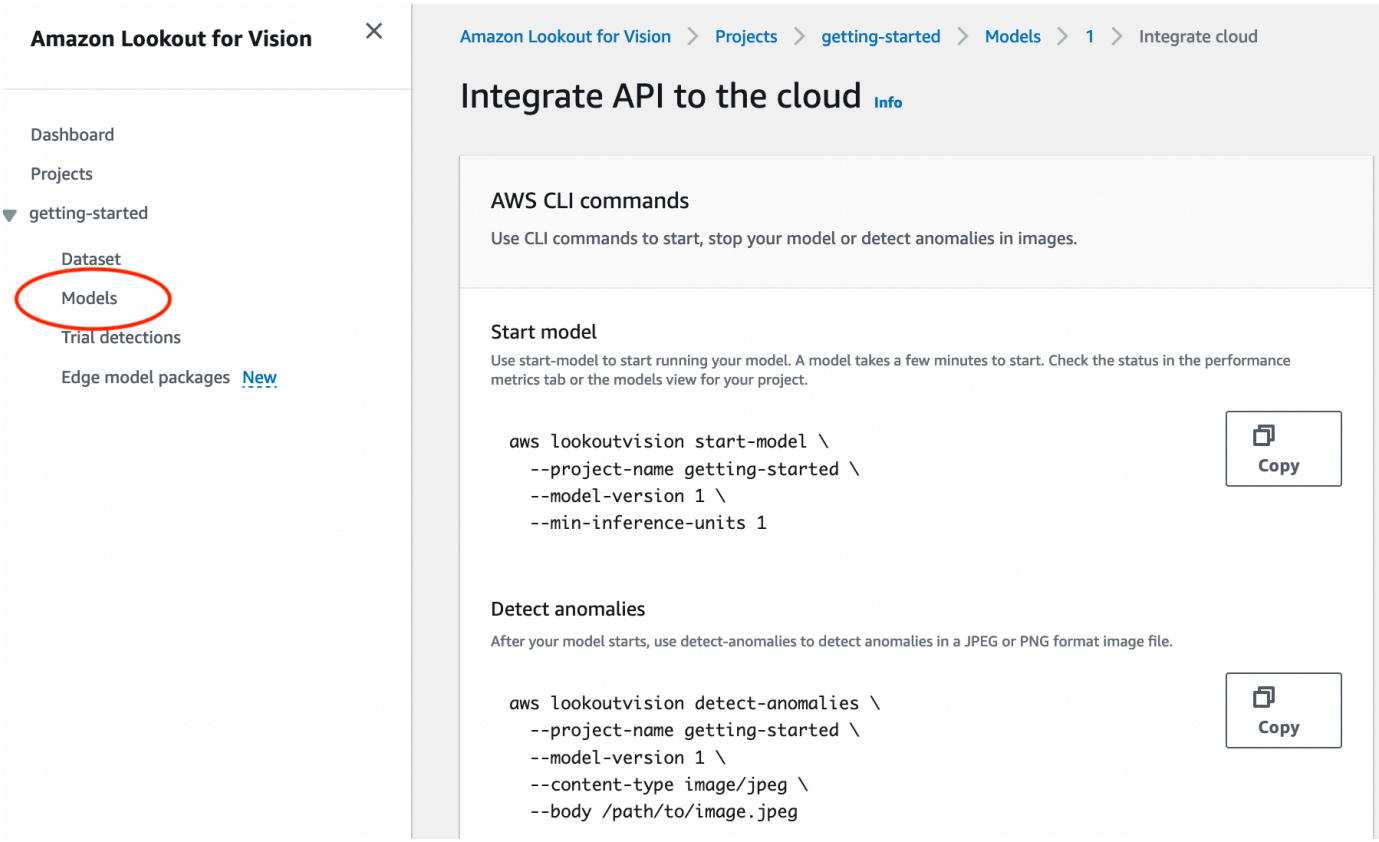
```
aws lookoutvision start-model \
  --project-name getting-started \
  --model-version 1 \
  --min-inference-units 1 \
```

```
  --profile lookoutvision-access
```

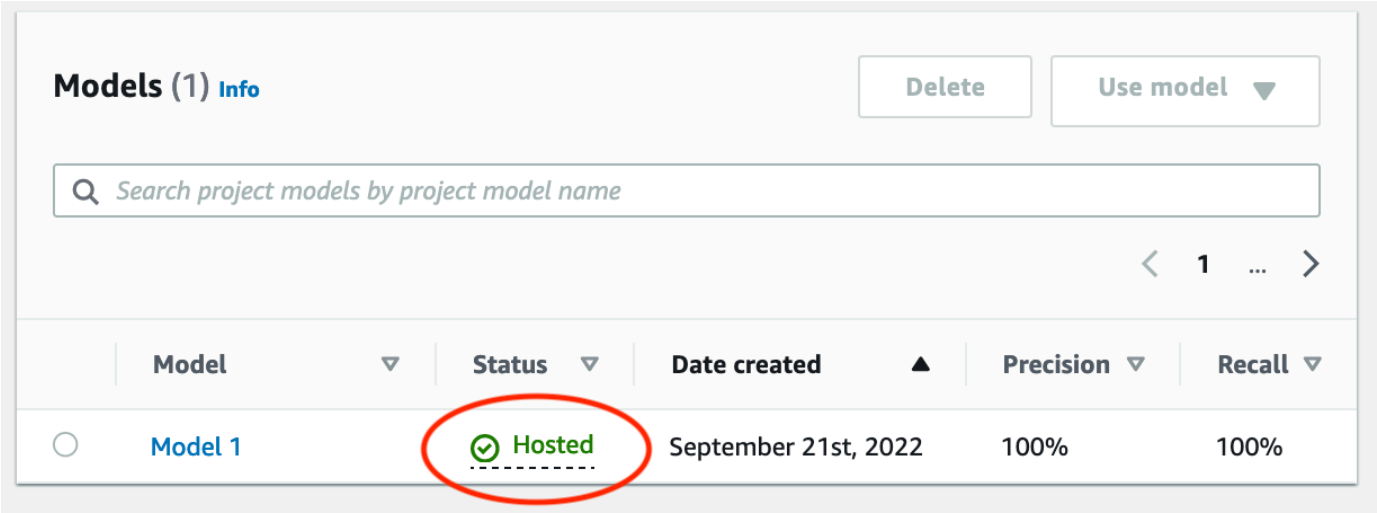
If the call is successful, the following output is displayed:

```
{
    "Status": "STARTING_HOSTING"
}
```

5. Back in the console, choose **Models** in the navigation pane.



6. Wait until the status of the model (Model 1) in the **Status** column displays **Hosted**. If you've previously trained a model in the project, wait for the latest model version to complete.

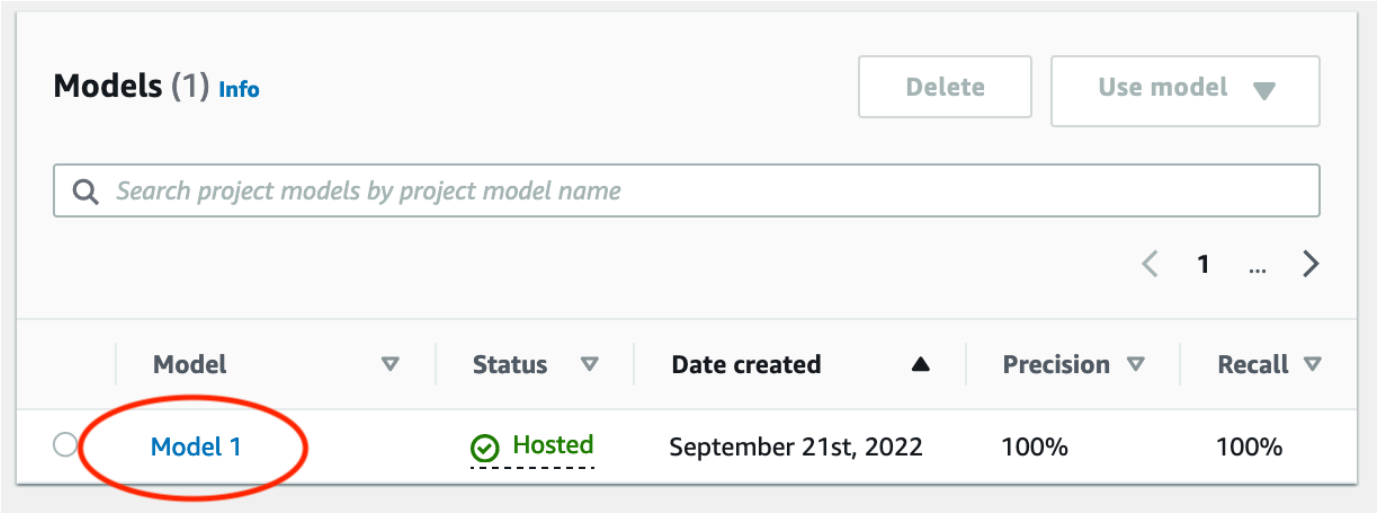


Step 4: Analyze an image

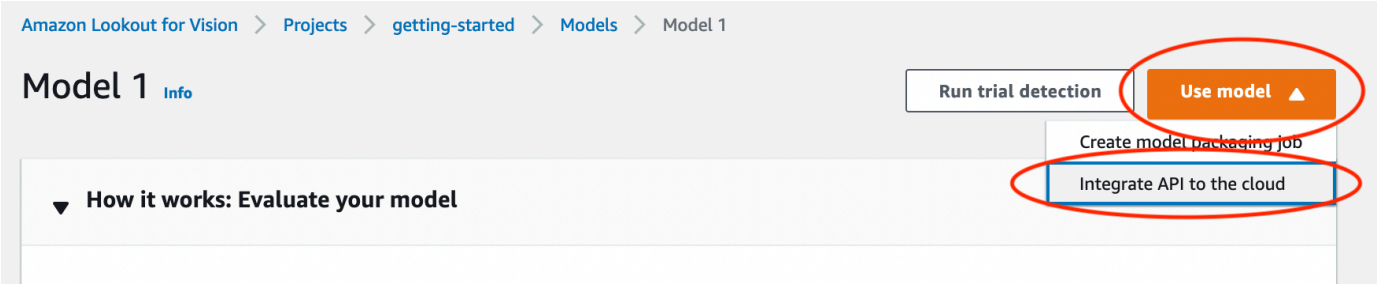
In this step, you analyze an image with your model. We provide example images that you can use in the getting started `test-images` folder in the Lookout for Vision documentation repository on your computer. For more information, see Detecting anomalies in an image.

To analyze an image

1. On the **Models** page, choose the model **Model 1**.



2. On the model's details page, choose **Use model** and then choose **Integrate API to the cloud**.



3. In the **AWS CLI commands** section, copy the `detect-anomalies` AWS CLI command.

4. At the command prompt, analyze an anomalous image by entering the `detect-anomalies` command from the previous step. For the `--body` parameter, specify an anomalous image from the getting started `test-images` folder on your computer. If you are using the `lookoutvision` profile to get credentials, add the `--profile lookoutvision-access` parameter. For example:

```
aws lookoutvision detect-anomalies \
  --project-name getting-started \
  --model-version 1 \
  --content-type image/jpeg \
  --body /path/to/test-images/test-anomaly-1.jpg \
  --profile lookoutvision-access
```

The output should look similar to the following:

```
{
    "DetectAnomalyResult": {
        "Source": {
            "Type": "direct"
        },
        "IsAnomalous": true,
        "Confidence": 0.983975887298584,
        "Anomalies": [
            {
```

```
                "Name": "background",
                "PixelAnomaly": {
                    "TotalPercentageArea": 0.9818974137306213,
                    "Color": "#FFFFFF"
                }
            },
            {
                "Name": "cracked",
                "PixelAnomaly": {
                    "TotalPercentageArea": 0.018102575093507767,
                    "Color": "#23A436"
                }
            }
        ],
        "AnomalyMask": "iVBORw0KGgoAAAANSUhEUgAAAkAAAAMACA......"
    }
}
```

5. In the output, note the following:

   - `IsAnomalous` is a Boolean for the predicted classification. `true` if the image is anomalous, otherwise `false`.
   - `Confidence` is a float value representing the confidence that Amazon Lookout for Vision has in the prediction. 0 is the lowest confidence, 1 is the highest confidence.
   - `Anomalies` is a list of anomalies found in the image. `Name` is the anomaly label. `PixelAnomaly` includes the total percentage area of the anomaly (`TotalPercentageArea`) and a color (`Color`) for the anomaly label. The list also includes a "background" anomaly that covers the area outside of anomalies found on the image.
   - `AnomalyMask` is a mask image that shows the location of the anomalies on the analyzed image.

   You can use information in the response to display a blend of the analyzed image and anomaly mask, as shown in the following example. For example code, see Showing classification and segmentation information.

Classification:
Prediction: Anomalous
Confidence: 99.9%
Segmentation:
Anomaly: cracked. Area: 6.2%

6. At the command prompt, analyze a normal image from the getting started `test-images` folder. If you are using the `lookoutvision` profile to get credentials, add the `--profile lookoutvision-access` parameter. For example:

```
aws lookoutvision detect-anomalies \
  --project-name getting-started \
  --model-version 1 \
  --content-type image/jpeg \
  --body /path/to/test-images/test-normal-1.jpg \
  --profile lookoutvision-access
```

The output should look similar to the following:

```
{
    "DetectAnomalyResult": {
        "Source": {
            "Type": "direct"
        },
        "IsAnomalous": false,
        "Confidence": 0.9916400909423828,
        "Anomalies": [
            {
                "Name": "background",
                "PixelAnomaly": {
                    "TotalPercentageArea": 1.0,
                    "Color": "#FFFFFF"
                }
            }
        ],
        "AnomalyMask": "iVBORw0KGgoAAAANSUhEUgAAAkAAAA....."
    }
}
```

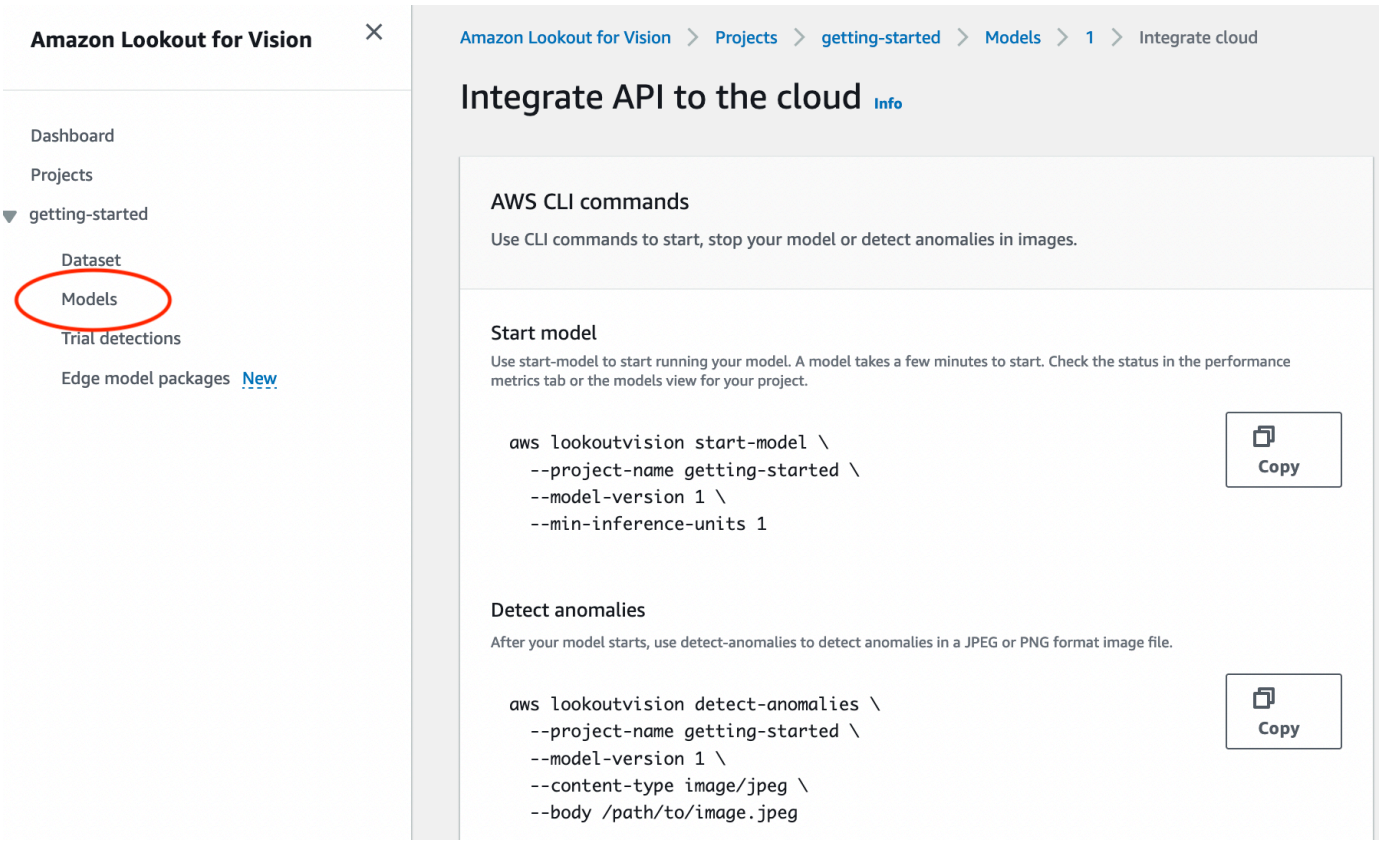
7. In the output, note that the `false` value for `IsAnomalous` classifies the image as having no anomalies. Use `Confidence` to help decide your confidence in the classification. Also, the `Anomalies` array only has the `background` anomaly label.

Step 5: Stop the model

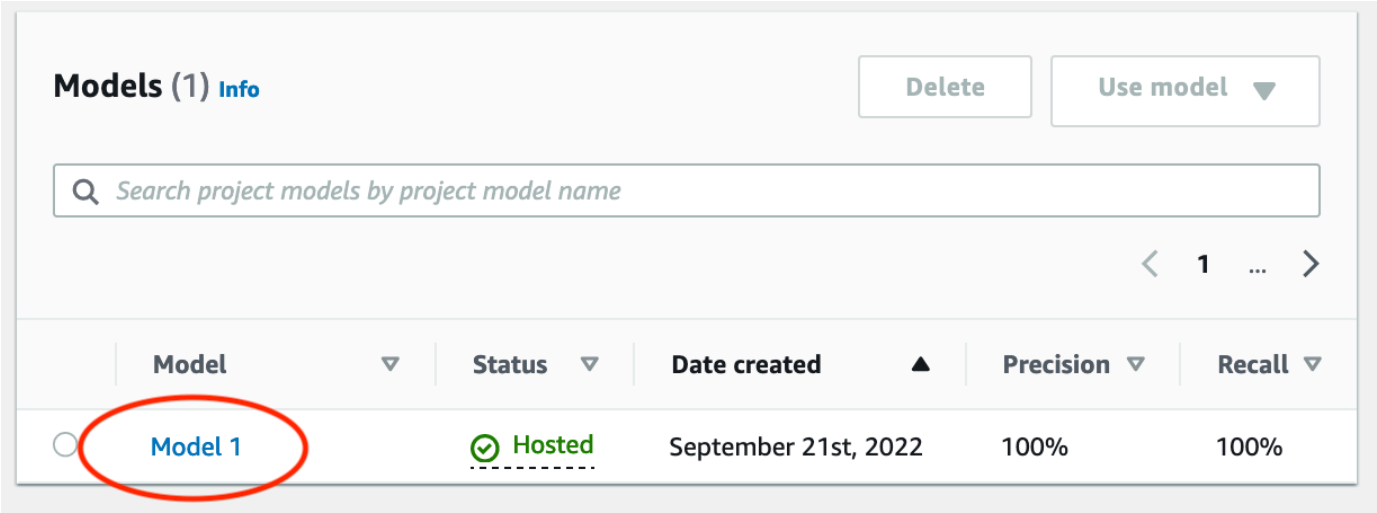
In this step, you stop hosting the model. You are charged for the amount of time your model is running. If you aren't using the model, you should stop it. You can restart the model when you next need it. For more information, see Starting your Amazon Lookout for Vision model.

To stop the model.

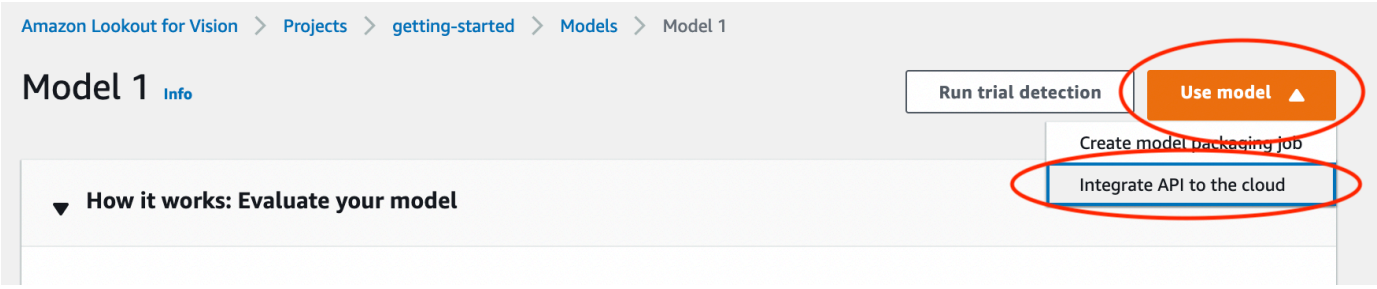
1. Choose **Models** in the navigation pane.



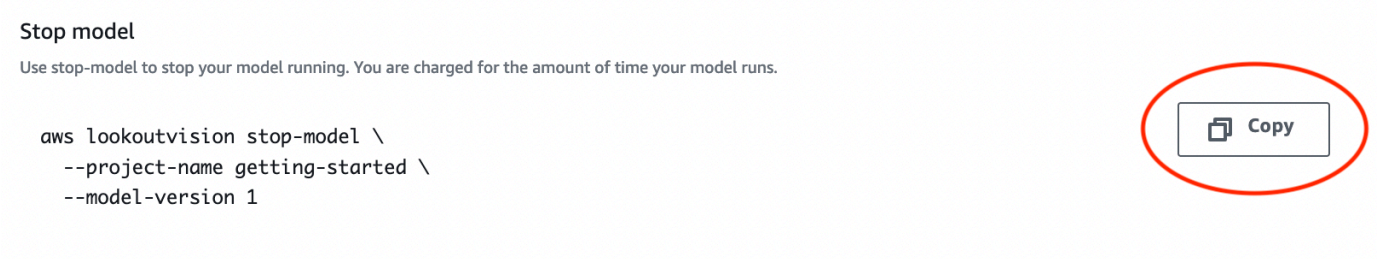
2. In the **Models** page, choose the model **Model 1**.



3. On the model's details page, choose **Use model** and then choose **Integrate API to the cloud**.



4. In the **AWS CLI commands** section, copy the `stop-model` AWS CLI command.



5. At the command prompt, stop the model by entering the `stop-model` AWS CLI command from the previous step. If you are using the `lookoutvision` profile to get credentials, add the `--profile lookoutvision-access` parameter. For example:

```
aws lookoutvision stop-model \
  --project-name getting-started \
  --model-version 1 \
  --profile lookoutvision-access
```

If the call is successful, the following output is displayed:

```
{
    "Status": "STOPPING_HOSTING"
}
```

6. Back in the console, choose **Models** in the left navigation page.
7. The model has stopped when the status of the model in the **Status** column is **Training complete**.

Next steps

When you are ready create a model with your own images, start by following the instructions in Creating your project. The instructions include steps for creating a model with the Amazon Lookout for Vision console and with the AWS SDK.

If you want to try other example datasets, see Example code and datasets.

Creating your Amazon Lookout for Vision model

An Amazon Lookout for Vision model is a machine learning model that predicts the presence of anomalies in new images by finding patterns in images used to train the model. This section shows you how to create and train a model. After you train your model, you evaluate its performance. For more information, see Improving your Amazon Lookout for Vision model.

Before you create your first model, we recommend that you read Understanding Amazon Lookout for Vision and Getting started with Amazon Lookout for Vision. If you are using the AWS SDK, read Call an Amazon Lookout for Vision operation.

Topics

- Creating your project
- Creating your dataset
- Labeling images
- Training your model
- Troubleshooting model training

Creating your project

An Amazon Lookout for Vision project is a grouping of the resources needed to create and manage a Lookout for Vision model. A project manages the following:

- **Dataset** – The images and image labels used to train a model. For more information, see Creating your dataset.
- **Model** – The software that you train to detect anomalies. You can have multiple versions of a model. For more information, see Training your model.

We recommend that you use a project for a single use case, such as detecting anomalies in a single type of machine part.

> **Note**
>
> You can use AWS CloudFormation to provision and configure Amazon Lookout for Vision projects. For more information, see Creating Amazon Lookout for Vision resources with AWS CloudFormation.

To view your projects, see Viewing your projects or open the Using the Amazon Lookout for Vision dashboard. To delete a model, see Deleting a model.

Topics

- Creating a project (console)
- Creating a project (SDK)

Creating a project (console)

The following procedure shows you how to create a project using the console.

To create a project (console)

1. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
2. In the left navigation pane, choose **Projects**.
3. Choose **Create project**.
4. In **Project name**, enter a name for your project.
5. Choose **Create project**. The details page for your project is displayed.
6. Follow the steps in Creating your dataset to create your dataset.

Creating a project (SDK)

You use the CreateProject operation to create an Amazon Lookout for Vision project. The response from `CreateProject` includes the project name and the Amazon Resource Name (ARN) of the project. Afterwards, call CreateDataset to add a training and a test dataset to your project. For more information, see Creating a dataset with a manifest file (SDK).

To view the projects that you have created in a project, call `ListProjects`. For more information, see Viewing your projects.

To create a project (SDK)

1. If you haven't already done so, install and configure the AWS CLI and the AWS SDKs. For more information, see Step 4: Set up the AWS CLI and AWS SDKs.
2. Use the following example code to create a model.

   CLI

   Change the value of `project-name` to the name that you want to use for the project.

```
aws lookoutvision create-project --project-name project name \
  --profile lookoutvision-access
```

   Python

   This code is taken from the AWS Documentation SDK examples GitHub repository. See the full example here.

```
    @staticmethod
    def create_project(lookoutvision_client, project_name):
        """
        Creates a new Lookout for Vision project.

        :param lookoutvision_client: A Boto3 Lookout for Vision client.
        :param  project_name: The name for the new project.
        :return project_arn: The ARN of the new project.
        """
        try:
            logger.info("Creating project: %s", project_name)
            response =
 lookoutvision_client.create_project(ProjectName=project_name)
            project_arn = response["ProjectMetadata"]["ProjectArn"]
            logger.info("project ARN: %s", project_arn)
        except ClientError:
            logger.exception("Couldn't create project %s.", project_name)
            raise
        else:
            return project_arn
```

Java V2

This code is taken from the AWS Documentation SDK examples GitHub repository. See the full example here.

```
/**
 * Creates an Amazon Lookout for Vision project.
 *
 * @param lfvClient   An Amazon Lookout for Vision client.
 * @param projectName The name of the project that you want to create.
 * @return ProjectMetadata Metadata information about the created project.
 */
public static ProjectMetadata createProject(LookoutVisionClient lfvClient,
 String projectName)
                throws LookoutVisionException {

        logger.log(Level.INFO, "Creating project: {0}", projectName);
        CreateProjectRequest createProjectRequest =
 CreateProjectRequest.builder().projectName(projectName)
                        .build();

        CreateProjectResponse response =
 lfvClient.createProject(createProjectRequest);

        logger.log(Level.INFO, "Project created. ARN: {0}",
 response.projectMetadata().projectArn());

        return response.projectMetadata();

}
```

3. Follow the steps in Creating a dataset using an Amazon SageMaker AI Ground Truth manifest file to create your dataset.

Creating your dataset

A dataset contains the images and assigned labels that you use to train and test a model. You create the dataset for your project with the Amazon Lookout for Vision console or with the

CreateDataset operation. The dataset images must be labeled according to the type of model that you want to create (image classification or image segmentation).

Topics

- Preparing images for a dataset
- Creating the dataset
- Creating a dataset using images stored on your local computer
- Creating a dataset using images stored in an Amazon S3 bucket
- Creating a dataset using an Amazon SageMaker AI Ground Truth manifest file

Preparing images for a dataset

You need a collection of images to create a dataset. Your images must be PNG or JPEG format files. The number and type of images you need depends on if your project has a single a single dataset or separate training and test datasets.

Single dataset project

To create an image classification model, you need the following to start training:

- At least 20 images of normal objects.
- At least 10 images of anomalous objects.

To create an image segmentation model, you need the following to start training:

- At least 20 images of each anomaly type.
- Each anomalous image (image with anomaly types present) must have only one type of anomaly.
- At least 20 images of normal objects.

Separate training and test dataset project

To create an image classification model, you need the following:

- At least 10 images of normal objects in the training dataset.
- At least 10 images of normal objects in the test dataset.

- At least 10 images of anomalous objects in the test dataset.

To create an image segmentation model, you need the following:

- Each dataset needs at least 10 images of each anomaly type.
- Each anomalous image (image with anomaly types present) must contain only one type of anomaly.
- Each dataset must have at least 10 images of normal objects.

To create a higher quality model, use more than the minimum number of images. If you are creating a segmentation model, we recommend including images with multiple anomaly types, but these don't count towards the minimum that Lookout for Vision needs to start training.

Your images should be of a single type of object. Also, you should have consistent image capture conditions, such as camera positioning, lighting, and object pose.

All images in the training and test datasets must have the same dimensions. Later, the images that you analyze with your trained model must have the same dimensions as the training and test dataset images. For more information, see *Detecting anomalies in an image*.

All training and test images must be unique images, preferably of unique objects. Normal images should capture the normal variations of the object being analyzed. Anomalous images should capture a diverse sampling of anomalies.

Amazon Lookout for Vision provides example images that you can use. For more information, see Image classification dataset.

For image limits, see *Quotas*.

Creating the dataset

When you create the dataset for your project, you choose the initial dataset configuration of your project. You also choose where Lookout for Vision imports the images from.

Choosing a dataset configuration for your project

When you create the first dataset in your project, you choose one of the following dataset configurations:

- **Single dataset** – A single dataset project uses a single dataset to train and test your model. Using a single dataset simplifies training by letting Amazon Lookout for Vision choose the training and test images. During training, Amazon Lookout for Vision, internally splits the dataset into a training dataset and a test dataset. You don't have access to the split datasets. We recommend using a single dataset project for most scenarios.
- **Separate training and test datasets** – If you want finer control over training, testing, and performance tuning, you can configure your project to have separate training and test datasets. Use a separate test dataset if you want control over the images used for testing, or if you already have a benchmark set of images that you want to use.

You can add a test dataset to an existing single dataset project. The single dataset then becomes the training dataset. If you remove the test dataset from a project with separate training and test datasets, the project becomes a single dataset project. For more information, see Deleting a dataset.

Importing images

When you create a dataset, you choose where to import the images from. Depending on how you import the images, the images might already be labeled. If the images aren't labeled after creating the dataset, see Labeling images.

You create a dataset and import its images in one of the following ways:

- Import images from your local computer. The images aren't labeled. You add or labels by using the Lookout for Vision console.
- Import images from an S3 bucket. Amazon Lookout for Vision can classify images by using the folder names to label the images. Use `normal` for normal images. Use `anomaly` for anomalous images. You can't automatically assign segmentation labels.
- Import an Amazon SageMaker AI Ground Truth manifest file, which includes labeled images. You can create and import your own manifest file. If you have many images, consider using the SageMaker AI Ground Truth labeling service. You then import the output manifest file from the Amazon SageMaker AI Ground Truth job. If necessary, you can use the Lookout for Vision console to add or change labels.

If you're using the AWS SDK, you create a dataset with an Amazon SageMaker AI Ground Truth manifest file. For more information, see Creating a dataset using an Amazon SageMaker AI Ground Truth manifest file.

If, after creating your dataset, your images are labeled, you can train the model. If the images aren't labeled, add the labels according to the type of model that you want to create. For more information, see Labeling images.

You can add more images to an existing dataset. For more information, see Adding images to your dataset.

Creating a dataset using images stored on your local computer

You can create a dataset by using images that are loaded directly from your computer. You can upload up to 30 images at a time. In this procedure, you can create a single dataset project, or a project with separate training and test datasets.

> **Note**
>
> If you've just completed Creating your project, the console should show your project dashboard and you don't need to do steps 1 - 3.

To create a dataset using images on a local computer (console)

1. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
2. In the left navigation pane, choose **Projects**.
3. In the **Projects** page, choose the project to which you want to add a dataset.
4. On the project details page, choose **Create dataset**.
5. Choose the **Single dataset** tab or the **Separate training and test datasets** tab and follow the steps.

   Single dataset

   a. In the **Dataset configuration** section, choose **Create a single dataset**.
   b. In the **Image source configuration** section, choose **Upload images from your computer**.
   c. Choose **Create dataset**.
   d. On the dataset page, choose **Add images**.
   e. Choose the images you want to upload into the dataset from your computer files. You can drag the images or choose the images that you want to upload from your local computer.

f. Choose **Upload images**.

Separate training and test datasets

a. In the **Dataset configuration** section, choose **Create a training dataset and a test dataset**.

b. In the **Training dataset details** section, choose **Upload images from your computer**.

c. In the **Test dataset details** section, choose **Upload images from your computer**.

> **Note**
>
> Your training and test datasets can have different image sources.

d. Choose **Create dataset**. A dataset page appears with a **Training** tab and a **Test** tab for the respective datasets.

e. Choose **Actions** and then choose **Add images to training dataset**.

f. Choose the images you want to upload to the dataset. You can drag the images or choose the images that you want to upload from your local computer.

g. Choose **Upload images**.

h. Repeat steps 5e - 5g. For step 5e, choose **Actions** and then choose **Add images to test dataset**.

6. Follow the steps in Labeling images to label your images.

7. Follow the steps in Training your model to train your model.

Creating a dataset using images stored in an Amazon S3 bucket

You can create a dataset using images stored in an Amazon S3 bucket. With this option, you can use the folder structure in your Amazon S3 bucket to automatically classify your images. You can store the images in the console bucket or another Amazon S3 bucket in your account.

Setting up folders for automatic labeling

During dataset creation, you can choose to assign label names to images based on the name of the folder that contains the images. The folders must be a child of the Amazon S3 folder path that you specify in **S3 URI** when you create the dataset.

The following is the `train` folder for the Getting Started example images. If you specify the Amazon S3 folder location as `S3-bucket/circuitboard/train/`, the images in the folder `normal` are assigned the label `Normal`. Images in the folder `anomaly` are assigned the label `Anomaly`. The names of deeper child folders aren't used to label images.

```
S3-bucket
  ### circuitboard
    ### train
        ### anomaly
            ### train-anomaly_1.jpg
            ### train-anomaly_2.jpg
            ### .
            ### .
        ### normal
            ### train-normal_1.jpg
            ### train-normal_2.jpg
            ### .
            ### .
```

Creating a dataset using images from an Amazon S3 bucket

The following procedure creates a dataset using the classification example images stored in an Amazon S3 bucket. To use your own images, create the folder structure described in Setting up folders for automatic labeling.

The procedure also shows how to create a single dataset project, or a project that uses separate training and test datasets.

If you don't choose to automatically label your images, you need to label the images after the datasets is created. For more information, see Classifying images (console).

> **Note**
>
> If you've just completed Creating your project, the console should show your project dashboard and you don't need to do steps 1 - 4.

To create a dataset using images stored in an Amazon S3 bucket

1. If you haven't already done so, upload the getting started images to your Amazon S3 bucket. For more information, see Image classification dataset.
2. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
3. In the left navigation pane, choose **Projects**.
4. In the **Projects** page, choose the project to which you want to add a dataset. The details page for your project is displayed.
5. Choose **Create dataset**. The **Create dataset** page is shown.

   > **Tip**
   >
   > If you're following the Getting Started instructions, choose **Create a training dataset and a test dataset**.

6. Choose the **Single dataset** tab or the **Separate training and test datasets** tab and follow the steps.

   Single dataset

   a. In the **Dataset configuration** section, choose **Create a single dataset**.
   b. Enter the information for steps 7 - 9 in the **Image source configuration** section.

   Separate training and test datasets

   a. In the **Dataset configuration** section, choose **Create a training dataset and a test dataset**.
   b. For your training dataset, enter the information for steps 7 - 9 in the **Training dataset details** section.
   c. For your test dataset, enter the information for steps 7 - 9 in the **Test dataset details** section.

   > **Note**
   >
   > Your training and test datasets can have different image sources.

7. Choose **Import images from Amazon S3 bucket**.
8. In **S3 URI**, enter the Amazon S3 bucket location and folder path. Change `bucket` to the name of your Amazon S3 bucket.

   a. If you're creating a single dataset project or a training dataset, enter the following:

   ```
   s3://bucket/circuitboard/train/
   ```

   b. If you're creating a test dataset enter the following:

   ```
   s3://bucket/circuitboard/test/
   ```

9. Choose **Automatically attach labels to images based on the folder**.
10. Choose **Create dataset**. A dataset page opens with your labeled images.
11. Follow the steps in Training your model to train your model.

Creating a dataset using an Amazon SageMaker AI Ground Truth manifest file

A manifest file contains information about the images and image labels that you can use to train and test a model. You can store a manifest file in an Amazon S3 bucket and use it to create a dataset. You can create your own manifest file or you can use an existing manifest file, such as the output from an Amazon SageMaker AI Ground Truth job.

Topics

- Using an Amazon Sagemaker Ground Truth job
- Creating a manifest file

Using an Amazon Sagemaker Ground Truth job

Labeling images can take significant time. For example, it can take 10s of seconds to accurately draw a mask around an anomaly. If you have 100s of images, it might take several hours to label them. As an alternative to labeling the images yourself, consider using Amazon SageMaker Ground Truth.

With Amazon SageMaker AI Ground Truth, you can use workers from either Amazon Mechanical Turk a vendor company that you choose, or an internal, private workforce to create a labeled set of images. For more information, see Use Amazon SageMaker AI Ground Truth to Label Data.

There is a cost for using Amazon Mechanical Turk. Also, It might take several days to complete an Amazon Ground Truth labeling job. If cost is an issue, or if you need to train your model quickly, we recommend that you use the Amazon Lookout for Vision console to label your images.

You can use an Amazon SageMaker AI Ground Truth labeling job to label images suitable for images classification models and image segmentation models. After the job completes, you use the output manifest file to create an Amazon Lookout for Vision dataset.

Image classification

To label images for an image classification model, create a labeling job for an Image Classification (Single Label) task.

Image segmentation

To label images for an image segmentation model, create a labeling job for an Image Classification (Single Label) task. Then, chain the job to create a labeling job for an Image Semantic Segmentation task.

You can also use a labeling job to create a partial manifest file for an image segmentation model. For example, you can classify images with an Image Classification (Single Label) task. After creating a Lookout for Vision dataset with the job output, use the Amazon Lookout for Vision console to add segmentation masks and anomaly labels to the dataset images.

Labeling images with Amazon SageMaker AI Ground Truth

The following procedure shows how to label images with Amazon SageMaker AI Ground Truth image labeling tasks. The procedure creates an image classification manifest file and optionally chains the image labeling task to create an image segmentation manifest file. If you want your project to have a separate test dataset, repeat this procedure to create the manifest file for the test dataset.

To label images with Amazon SageMaker AI Ground Truth (Console)

1. Create a Ground Truth job for an *Image Classification (Single Label)* task by following the instructions at Create a Labeling Job (Console).

   a. For step 10, choose **Image** from the **Task category** dropdown menu, and choose **Image Classification (Single Label)** as the task type.
   b. For step 16, in the **Image classification (Single Label) labeling tool** section, add two labels: **normal** and **anomaly**.
2. Wait until the workforce finishes classifying your images.
3. If you are creating a dataset for an image segmentation model, do the following. Otherwise go to to step 4.
   a. In the Amazon SageMaker AI Ground Truth console, open the **Labeling jobs** page.
   b. Choose the job you previously created. This enables the **Actions** menu.
   c. From the **Actions** menu, choose **Chain**. The job details page opens.
   d. In **task type**, choose **semantic segmentation**.
   e. Choose **Next**.
   f. In the **Semantic segmentation labeling tool** section, add anomaly labels for each type of anomaly that you want your model to find.
   g. Choose **Create**.
   h. Wait until the workforce labels your images.
4. Open the Ground Truth console and open the **Labeling jobs** page.
5. If you are creating an image classification model, choose the job you created in step 1. If you are creating an image segmentation model, choose the job created in step 3.
6. In **Labeling job summary** open the S3 location in **Output dataset location**. Note the manifest file location, which should be `s3://output-dataset-location/manifests/output/output.manifest`.
7. Repeat this procedure if you want to create a manifest file for a test dataset. Otherwise, follow the instructions at Creating the dataset to create a dataset with the manifest file.

Creating the dataset

Use this procedure to create a dataset in a Lookout for Vision project with the manifest file that you noted in step 6 of Labeling images with Amazon SageMaker AI Ground Truth. The manifest file creates the training dataset for a single dataset project. If you want your project to have a separate test dataset, you can run another Amazon SageMaker AI Ground Truth job to create a manifest file for the test dataset. Or you can create the manifest file yourself. You can also import images to

your test dataset from an Amazon S3 bucket or from your local computer. (The images might need labeling before you can train the model).

This procedure assumes that your project doesn't have any datasets.

To create a dataset with Lookout for Vision (Console)

1. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
2. Choose **Get started**.
3. In the left navigation pane, choose **Projects**.
4. Choose the project that you want to add to use with the manifest file.
5. In the **How it works** section, choose **Create dataset**.
6. Choose the **Single dataset** tab or the **Separate training and test datasets** tab and follow the steps.

   Single dataset

   1. Choose **Create a single dataset**.
   2. In the **Image source configuration** section, choose **Import images labeled by SageMaker Ground Truth**.
   3. For **.manifest file location**, enter the location of the manifest file you noted in step 6 of Labeling images with Amazon SageMaker AI Ground Truth.

   Separate training and test datasets

   1. Choose **Create a training dataset and a test dataset**.
   2. In the **Training dataset details** section, choose **Import images labeled by SageMaker Ground Truth**.
   3. In **.manifest file location**, the location of the manifest file you noted in step 6 of Labeling images with Amazon SageMaker AI Ground Truth.
   4. In the **Test dataset details** section, choose **Import images labeled by SageMaker Ground Truth**.
   5. In **.manifest file location**, the location of the manifest file you noted in step 6 of Labeling images with Amazon SageMaker AI Ground Truth. Remember that you need a separate manifest file for the test dataset.

7. Choose **Submit**.
8. Follow the steps in Training your model to train your model.

Creating a manifest file

You can create a dataset by importing an SageMaker AI Ground Truth format manifest file. If your images are labeled in a format that isn't a SageMaker AI Ground Truth manifest file, use the following information to create an SageMaker AI Ground Truth format manifest file.

Manifest files are in JSON lines format where each line is a complete JSON object representing the labeling information for an image. There are different formats for image classification and image segmentation. Manifest files must be encoded using UTF-8 encoding.

> **Note**
>
> The JSON line examples in this section are formatted for readability.

The images referenced by a manifest file must be located in the same Amazon S3 bucket. The manifest file can be in a different bucket. You specify the location of an image in the `source-ref` field of a JSON line.

You can create a manifest file by using code. The Amazon Lookout for Vision Lab Python Notebook shows how to create an image classification manifest file for the circuitboard example images. Alternatively, you can use the Datasets example code in the AWS Code Examples Repository.
You can easily create a manifest file by using a Comma Separated Values (CSV) file. For more information, see Creating a classification manifest file from a CSV file.

Topics

- Defining JSON lines for image classification
- Defining JSON lines for image segmentation
- Creating a classification manifest file from a CSV file
- Creating a dataset with a manifest file (console)
- Creating a dataset with a manifest file (SDK)

Defining JSON lines for image classification

You define a JSON line for each image that you want to use in an Amazon Lookout for Vision manifest file. If you want to create a classification model, the JSON line must include an image classification that is either normal or an anomaly. A JSON line is in SageMaker AI Ground Truth Classification Job Output format. A manifest file is made of one or more JSON lines, one for each image that you want to import.

To create a manifest file for classified images

1. Create an empty text file.
2. Add a JSON line for each image the that you want to import. Each JSON line should look similar to the following:

```
{"source-ref":"s3://lookoutvision-console-us-east-1-nnnnnnnnnn/gt-job/manifest/
IMG_1133.png","anomaly-label":1,"anomaly-label-metadata":{"confidence":0.95,"job-
name":"labeling-job/testclconsolebucket","class-name":"normal","human-
annotated":"yes","creation-date":"2020-04-15T20:17:23.433061","type":"groundtruth/
image-classification"}}
```

3. Save the file.

> **Note**
>
> You can use the extension `.manifest`, but it is not required.

4. Create a dataset using the manifest file that you created. For more information, see Creating a manifest file.

Classification JSON lines

In this section, you learn how to create a JSON line that classifies an image as normal or anomalous.

Anomaly JSON line

The following JSON line shows an image that is labeled as an anomaly. Note that the value of `class-name` is `anomaly`.

```
{
    "source-ref": "s3: //bucket/image/anomaly/abnormal-1.jpg",
    "anomaly-label-metadata": {
        "confidence": 1,
        "job-name": "labeling-job/auto-label",
        "class-name": "anomaly",
        "human-annotated": "yes",
        "creation-date": "2020-11-10T03:37:09.600",
        "type": "groundtruth/image-classification"
    },
    "anomaly-label": 1
}
```

Normal JSON line

The following JSON line shows an image labeled as normal. Note that the value of `class-name` is `normal`.

```
{
    "source-ref": "s3: //bucket/image/normal/2020-10-20_12-14-55_613.jpeg",
    "anomaly-label-metadata": {
        "confidence": 1,
        "job-name": "labeling-job/auto-label",
        "class-name": "normal",
        "human-annotated": "yes",
        "creation-date": "2020-11-10T03:37:09.603",
        "type": "groundtruth/image-classification"
    },
    "anomaly-label": 0
}
```

JSON line keys and values

The following information describes the keys and values in an Amazon Lookout for Vision JSON line.

source-ref

(Required) The Amazon S3 location of the image. The format is "`s3://BUCKET/OBJECT_PATH`". Images in an imported dataset must be stored in the same Amazon S3 bucket.

anomaly-label

(Required) The label attribute. Use the key `anomaly-label`, or another key name that you choose. The key value (`0` in the preceding example) is required by Amazon Lookout for Vision, but it isn't used. The output manifest created by Amazon Lookout for Vision converts the value to 1 for an anomalous image and a value of `0` for a normal image. The value of `class-name` determines if the image is normal or anomalous.

There must be corresponding metadata identified by the field name with *-metadata* appended. For example, `"anomaly-label-metadata"`.

anomaly-label-metadata

(Required) Metadata about the label attribute. The field name must be the same as the label attribute with *-metadata* appended.

confidence

(Optional) Currently not used by Amazon Lookout for Vision. If you do specify a value, use a value of 1.

job-name

(Optional) A name that you choose for the job that processes the image.

class-name

(Required) If the image contains normal content, specify `normal`, otherwise specify `anomaly`. If the value of `class-name` is any other value, the image is added to the dataset as an unlabeled image. To label an image, see Adding images to your dataset.

human-annotated

(Required) Specify `"yes"`, if the annotation was completed by a human. Otherwise, specify `"no"`.

creation-date

(Optional) The Coordinated Universal Time (UTC) date and time that the label was created.

type

(Required) The type of processing that should be applied to the image. For image-level anomaly labels, the value is `"groundtruth/image-classification"`.

Defining JSON lines for image segmentation

You define a JSON line for each image that you want to use in an Amazon Lookout for Vision manifest file. If you want to create a segmentation model, The JSON line must include segmentation and classification information for the image. A manifest file is made of one or more JSON lines, one for each image that you want to import.

To create a manifest file for segmented images

1. Create an empty text file.
2. Add a JSON line for each image the that you want to import. Each JSON line should look similar to the following:

```
{"source-ref":"s3://path-to-image","anomaly-label":1,"anomaly-label-metadata":
{"class-name":"anomaly","creation-date":"2021-10-12T14:16:45.668","human-
annotated":"yes","job-name":"labeling-job/classification-job","type":"groundtruth/
image-classification","confidence":1},"anomaly-mask-ref":"s3://path-to-
image","anomaly-mask-ref-metadata":{"internal-color-map":{"0":{"class-
name":"BACKGROUND","hex-color":"#ffffff","confidence":0.0},"1":{"class-
name":"scratch","hex-color":"#2ca02c","confidence":0.0},"2":{"class-
name":"dent","hex-color":"#1f77b4","confidence":0.0}},"type":"groundtruth/
semantic-segmentation","human-annotated":"yes","creation-
date":"2021-11-23T20:31:57.758889","job-name":"labeling-job/segmentation-job"}}
```

3. Save the file.

> **Note**
>
> You can use the extension `.manifest`, but it is not required.

4. Create a dataset using the manifest file that you created. For more information, see Creating a manifest file.

Segmentation JSON lines

In this section, you learn how to create a JSON line that includes segmentation and classification information for an image.

The following JSON line shows an image with segmentation and classification information. `anomaly-label-metadata` contains classification information. `anomaly-mask-ref` and `anomaly-mask-ref-metadata` contain segmentation information.

```
{
    "source-ref": "s3://path-to-image",
    "anomaly-label": 1,
    "anomaly-label-metadata": {
        "class-name": "anomaly",
        "creation-date": "2021-10-12T14:16:45.668",
        "human-annotated": "yes",
        "job-name": "labeling-job/classification-job",
        "type": "groundtruth/image-classification",
        "confidence": 1
    },
    "anomaly-mask-ref": "s3://path-to-image",
    "anomaly-mask-ref-metadata": {
        "internal-color-map": {
            "0": {
                "class-name": "BACKGROUND",
                "hex-color": "#ffffff",
                "confidence": 0.0
            },
            "1": {
                "class-name": "scratch",
                "hex-color": "#2ca02c",
                "confidence": 0.0
            },
            "2": {
                "class-name": "dent",
                "hex-color": "#1f77b4",
                "confidence": 0.0
            }
        },
        "type": "groundtruth/semantic-segmentation",
        "human-annotated": "yes",
        "creation-date": "2021-11-23T20:31:57.758889",
        "job-name": "labeling-job/segmentation-job"
    }
}
```

JSON line keys and values

The following information describes the keys and values in an Amazon Lookout for Vision JSON line.

source-ref

(Required) The Amazon S3 location of the image. The format is `"s3://BUCKET/OBJECT_PATH"`. Images in an imported dataset must be stored in the same Amazon S3 bucket.

anomaly-label

(Required) The label attribute. Use the key `anomaly-label`, or another key name that you choose. The key value (1 in the preceding example) is required by Amazon Lookout for Vision, but it isn't used. The output manifest created by Amazon Lookout for Vision converts the value to 1 for an anomalous image and a value of `0` for a normal image. The value of `class-name` determines if the image is normal or anomalous.

There must be corresponding metadata identified by the field name with *-metadata* appended. For example, `"anomaly-label-metadata"`.

anomaly-label-metadata

(Required) Metadata about the label attribute. Contains classification information. The field name must be the same as the label attribute with *-metadata* appended.

confidence

(Optional) Currently not used by Amazon Lookout for Vision. If you do specify a value, use a value of 1.

job-name

(Optional) A name that you choose for the job that processes the image.

class-name

(Required) If the image contains normal content, specify `normal`, otherwise specify `anomaly`. If the value of `class-name` is any other value, the image is added to the dataset as an unlabeled image. To label an image, see Adding images to your dataset.

human-annotated

(Required) Specify `"yes"`, if the annotation was completed by a human. Otherwise, specify `"no"`.

creation-date

(Optional) The Coordinated Universal Time (UTC) date and time that the label was created.

type

(Required) The type of processing that should be applied to the image. Use the value `"groundtruth/image-classification"`.

anomaly-mask-ref

(Required) The Amazon S3 location of the mask image. Use `anomaly-mask-ref` for the key name or use a key name of your choosing. The key must end with `-ref`. The mask image must contain colored masks for each anomaly type `internal-color-map`. The format is `"s3://BUCKET/OBJECT_PATH"`. Images in an imported dataset must be stored in the same Amazon S3 bucket. The mask image must be a Portable Network Graphic (PNG) format image.

anomaly-mask-ref-metadata

(Required) Segmentation metadata for the image. Use `anomaly-mask-ref-metadata` for the key name or use a key name of your choosing. The key name must end with `-ref-metadata`.

internal-color-map

(Required) A map of colors that map to individual anomaly types. The colors must match the colors in the mask image (`anomaly-mask-ref`).

key

(Required) The key into the map. The entry `0` must contain the class-name BACKGROUND that represents areas outside of anomalies on the image.

class-name

(Required) The name of the anomaly type, such as scratch or dent.

hex-color

(Required) The hex color for the anomaly type, such as `#2ca02c`. The color must match a color in `anomaly-mask-ref`. The value for the BACKGROUND anomaly type is always `#ffffff`.

confidence

(Required) Currently not used by Amazon Lookout for Vision, but a float value is required.

human-annotated

(Required) Specify `"yes"`, if the annotation was completed by a human. Otherwise, specify `"no"`.

creation-date

(Optional) The Coordinated Universal Time (UTC) date and time that the segmentation information was created.

type

(Required) The type of processing that should be applied to the image. Use the value `"groundtruth/semantic-segmentation"`.

Creating a classification manifest file from a CSV file

This example Python script simplifies the creation of a classification manifest file by using a Comma Separated Values (CSV) file to label images. You create the CSV file.

A manifest file describes the images used to train a model. A manifest file is made up of one or more JSON lines. Each JSON line describes a single image. For more information, see Defining JSON lines for image classification.

A CSV file represents tabular data over multiple rows in a text file. Fields on a row are separated by commas. For more information, see comma separated values. For this script, each row in your CSV file includes the S3 location of an image and the anomaly classification for the image (`normal` or `anomaly`). Each row maps to a JSON Line in the manifest file.

For example, The following CSV file describes some of the images in the example images.

```
s3://s3bucket/circuitboard/train/anomaly/train-anomaly_1.jpg,anomaly
s3://s3bucket/circuitboard/train/anomaly/train-anomaly_10.jpg,anomaly
s3://s3bucket/circuitboard/train/anomaly/train-anomaly_11.jpg,anomaly
s3://s3bucket/circuitboard/train/normal/train-normal_1.jpg,normal
s3://s3bucket/circuitboard/train/normal/train-normal_10.jpg,normal
s3://s3bucket/circuitboard/train/normal/train-normal_11.jpg,normal
```

The script generates JSON Lines for each row. For example, the following is the JSON Line for the first row (`s3://s3bucket/circuitboard/train/anomaly/train-anomaly_1.jpg,anomaly`) .

```
{"source-ref": "s3://s3bucket/csv_test/train_anomaly_1.jpg","anomaly-label":
 1,"anomaly-label-metadata": {"confidence": 1,"job-name": "labeling-job/anomaly-
classification","class-name": "anomaly","human-annotated": "yes","creation-date":
 "2022-02-04T22:47:07","type": "groundtruth/image-classification"}}
```

If your CSV file doesn't include the Amazon S3 path for the images, use the `--s3-path` command line argument to specify the Amazon S3 path to the images.

Before creating the manifest file, the script checks for duplicate images in the CSV file and any image classifications that are not `normal` or `anomaly`. If duplicates image or image classification errors are found, the script does the following:

- Records the first valid image entry for all images in a deduplicated CSV file.
- Records duplicate occurences of an image in the errors file.
- Records image classifications that are not `normal` or `anomaly` in the errors file.
- Doesn't create a manifest file.

The errors file includes the line number where a duplicate image or classification error is found in the input CSV file. Use the errors CSV file to update the input CSV file and then run the script again. Alternatively, use the errors CSV file to update the deduplicated CSV file, which only contains unique image entries and images with no image classification errors. Rerun the script with the updated deduplicated CSV file.

If no duplicates or errors are found in the input CSV file, the script deletes the deduplicated image CSV file and errors file, as they are empty.

In this procedure, you create the CSV file and run the Python script to create the manifest file. The script has been tested with Python version 3.7.

To create a manifest file from a CSV file

1. Create a CSV file with the following fields in each row (one row per image). Don't add a header row to the CSV file.

Field 1	Field 2
The image name or the Amazon S3 path the image. For example, `s3://s3bucket/`	The anomaly classification for the image (`normal` or `anomaly`).

Field 1	Field 2
`circuitboard/train/anomaly/train-anomaly_10.jpg` . You can't have a mixture of images with the Amazon S3 path and images without.	

For example `s3://s3bucket/circuitboard/train/anomaly/image_10.jpg,anomaly` or `image_11.jpg,normal`

2. Save the CSV file.
3. Run the following Python script. Supply the following arguments:
   - `csv_file` – The CSV file that you created in step 1.
   - (Optional)`--s3-path` *`s3://path_to_folder/`* – The Amazon S3 path to add to the image file names (field 1). Use `--s3-path` if the images in field 1 don't already contain an S3 path.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier:  Apache-2.0
"""
Purpose
Shows how to create an Amazon Lookout for Vision manifest file from a CSV file.
The CSV file format is image location,anomaly classification (normal or anomaly)
For example:
s3://s3bucket/circuitboard/train/anomaly/train_11.jpg,anomaly
s3://s3bucket/circuitboard/train/normal/train_1.jpg,normal

If necessary, use the bucket argument to specify the Amazon S3 bucket folder for
 the images.
"""

from datetime import datetime, timezone
import argparse
import logging
import csv
import os
import json

logger = logging.getLogger(__name__)
```

```
def check_errors(csv_file):
    """
    Checks for duplicate images and incorrect classifications in a CSV file.
    If duplicate images or invalid anomaly assignments are found, an errors CSV
 file
    and deduplicated CSV file are created. Only the first
    occurrence of a duplicate is recorded. Other duplicates are recorded in the
 errors file.
    :param csv_file: The source CSV file
    :return: True if errors or duplicates are found, otherwise false.
    """

    logger.info("Checking %s.", csv_file)

    errors_found = False
    errors_file = f"{os.path.splitext(csv_file)[0]}_errors.csv"
    deduplicated_file = f"{os.path.splitext(csv_file)[0]}_deduplicated.csv"

    with open(csv_file, 'r', encoding="UTF-8") as input_file,\
            open(deduplicated_file, 'w', encoding="UTF-8") as dedup,\
            open(errors_file, 'w', encoding="UTF-8") as errors:

        reader = csv.reader(input_file, delimiter=',')
        dedup_writer = csv.writer(dedup)
        error_writer = csv.writer(errors)
        line = 1
        entries = set()
        for row in reader:

            # Skip empty lines.
            if not ''.join(row).strip():
                continue

            # Record any incorrect classifications.
            if not row[1].lower() == "normal" and not row[1].lower() == "anomaly":
                error_writer.writerow(
                    [line, row[0], row[1], "INVALID_CLASSIFICATION"])
                errors_found = True

            # Write first image entry to dedup file and record duplicates.
            key = row[0]
            if key not in entries:
```

```
                dedup_writer.writerow(row)
                entries.add(key)
            else:
                error_writer.writerow([line, row[0], row[1], "DUPLICATE"])
                errors_found = True
            line += 1

    if errors_found:
        logger.info("Errors found check %s.", errors_file)
    else:
        os.remove(errors_file)
        os.remove(deduplicated_file)

    return errors_found


def create_manifest_file(csv_file, manifest_file, s3_path):
    """
    Read a CSV file and create an Amazon Lookout for Vision classification manifest
 file.
    :param csv_file: The source CSV file.
    :param manifest_file: The name of the manifest file to create.
    :param s3_path: The Amazon S3 path to the folder that contains the images.
    """
    logger.info("Processing CSV file %s.", csv_file)

    image_count = 0
    anomalous_count = 0

    with open(csv_file, newline='', encoding="UTF-8") as csvfile,\
        open(manifest_file, "w", encoding="UTF-8") as output_file:

        image_classifications = csv.reader(
            csvfile, delimiter=',', quotechar='|')

        # Process each row (image) in the CSV file.
        for row in image_classifications:
            # Skip empty lines.
            if not ''.join(row).strip():
                continue

            source_ref = str(s3_path) + row[0]
            classification = 0
```

```
            if row[1].lower() == 'anomaly':
                classification = 1
                anomalous_count += 1

           # Create the JSON line.
            json_line = {}
            json_line['source-ref'] = source_ref
            json_line['anomaly-label'] = str(classification)

            metadata = {}
            metadata['confidence'] = 1
            metadata['job-name'] = "labeling-job/anomaly-classification"
            metadata['class-name'] = row[1]
            metadata['human-annotated'] = "yes"
            metadata['creation-date'] = datetime.now(timezone.utc).strftime('%Y-%m-
%dT%H:%M:%S.%f')
            metadata['type'] = "groundtruth/image-classification"

            json_line['anomaly-label-metadata'] = metadata

            output_file.write(json.dumps(json_line))
            output_file.write('\n')
            image_count += 1

    logger.info("Finished creating manifest file %s.\n"
                "Images: %s\nAnomalous: %s",
                manifest_file,
                image_count,
                anomalous_count)
    return image_count, anomalous_count


def add_arguments(parser):
    """
    Add command line arguments to the parser.
    :param parser: The command line parser.
    """

    parser.add_argument(
        "csv_file", help="The CSV file that you want to process."
    )

    parser.add_argument(
        "--s3_path", help="The Amazon S3 bucket and folder path for the images."
```

```
        " If not supplied, column 1 is assumed to include the Amazon S3 path.",
 required=False
    )


def main():

    logging.basicConfig(level=logging.INFO,
                        format="%(levelname)s: %(message)s")

    try:

        # Get command line arguments.
        parser = argparse.ArgumentParser(usage=argparse.SUPPRESS)
        add_arguments(parser)
        args = parser.parse_args()
        s3_path = args.s3_path
        if s3_path is None:
            s3_path = ""

        csv_file = args.csv_file
        csv_file_no_extension = os.path.splitext(csv_file)[0]
        manifest_file = csv_file_no_extension + '.manifest'

        # Create manifest file if there are no duplicate images.
        if check_errors(csv_file):
            print(f"Issues found. Use {csv_file_no_extension}_errors.csv "\
                "to view duplicates and errors.")
            print(f"{csv_file}_deduplicated.csv contains the first"\
                "occurrence of a duplicate.\n"
                  "Update as necessary with the correct information.")
            print(f"Re-run the script with
 {csv_file_no_extension}_deduplicated.csv")
        else:
            print('No duplicates found. Creating manifest file.')

            image_count, anomalous_count = create_manifest_file(csv_file,
 manifest_file, s3_path)

            print(f"Finished creating manifest file: {manifest_file} \n")

            normal_count = image_count-anomalous_count
            print(f"Images processed: {image_count}")
            print(f"Normal: {normal_count}")
```

```
            print(f"Anomalous: {anomalous_count}")

    except FileNotFoundError as err:
        logger.exception("File not found.:%s", err)
        print(f"File not found: {err}. Check your input CSV file.")

if __name__ == "__main__":
    main()
```

4. If duplicate images occur or classification errors occur:

   a. Use the errors file to update the dedupulicated CSV file or the input CSV file.

   b. Run the script again with the updated deduplicated CSV file or updated input CSV file.

5. If you plan to use a test dataset, repeat steps 1–4 to create a manifest file for your test dataset.

6. If necessary, copy the images from your computer to the Amazon S3 bucket path that you specified in column 1 of the CSV file (or specified in the `--s3-path` command line). To copy the images, enter the following command at the command prompt.

```
aws s3 cp --recursive your-local-folder/ s3://your-target-S3-location/
```

7. Follow the instructions at Creating a dataset with a manifest file (console) to create a dataset. If you are use the AWS SDK, see Creating a dataset with a manifest file (SDK).

Creating a dataset with a manifest file (console)

The following procedure shows you how to create a training or test dataset by importing an SageMaker AI format manifest file that is stored in an Amazon S3 bucket.

After you create the dataset, you can add more images to the dataset, or add labels to images. For more information, see Adding images to your dataset.

To create a dataset using an SageMaker AI Ground Truth format manifest file (console)

1. Create, or use an existing, Amazon Lookout for Vision compatible SageMaker AI Ground Truth format manifest file. For more information, see Creating a manifest file.

2. Sign in to the AWS Management Console and open the Amazon S3 console at https://console.aws.amazon.com/s3/.

3. In an Amazon S3 bucket, create a folder to hold your manifest file.
4. Upload your manifest file to the folder that you just created.
5. In the Amazon S3 bucket, create a folder to store your images.
6. Upload your images to the folder you just created.

> **Important**
>
> The `source-ref` field value in each JSON line must map to images in the folder.

7. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
8. Choose **Get started**.
9. In the left navigation pane, choose **Projects**.
10. Choose the project that you want to add to use with the manifest file.
11. In the **How it works** section, choose **Create dataset**.
12. Choose the **Single dataset** tab or the **Separate training and test datasets** tab and follow the steps.

    Single dataset

    1. Choose **Create a single dataset**.
    2. In the **Image source configuration** section, choose **Import images labeled by SageMaker Ground Truth**.
    3. For **.manifest file location**, enter the location of your manifest file.

    Separate training and test datasets

    1. Choose **Create a training dataset and a test dataset**.
    2. In the **Training dataset details** section, choose **Import images labeled by SageMaker Ground Truth**.
    3. In **.manifest file location**, enter the location of your training manifest file.
    4. In the **Test dataset details** section, choose **Import images labeled by SageMaker Ground Truth**.
    5. In **.manifest file location**, enter the location of your test manifest file.

> **Note**
>
> Your training and test datasets can have different image sources.

13. Choose **Submit**.
14. Follow the steps in Training your model to train your model.

Amazon Lookout for Vision creates a dataset in the Amazon S3 bucket `datasets` folder. Your original `.manifest` file remains unchanged.

Creating a dataset with a manifest file (SDK)

You use the CreateDataset operation to create the datasets associated with an Amazon Lookout for Vision project.

If you want to use a single dataset for training and testing, create a single dataset with the `DatasetType` value set to `train`. During training, the dataset is internally split to make a training and test dataset. You don't have access to the split training and test datasets. If you want a separate test dataset, make a second call to `CreateDataset` with the `DatasetType` value set `test`. During training, the training and test datasets are used to train and test the model.

You can optionally use the `DatasetSource` parameter to specify the location of a SageMaker AI Ground Truth format manifest file that's used to populate the dataset. In this case, the call to `CreateDataset` is asynchronous. To check the current status, call `DescribeDataset`. For more information, see Viewing your datasets. If a validation error occurs during import, the value of `Status` is set to CREATE_FAILED and the status message (`StatusMessage`) is set.

> **Tip**
>
> If you are creating a dataset with the getting started example dataset, use the manifest file (`getting-started/dataset-files/manifests/train.manifest`) that the script creates in Step 1: Create the manifest file and upload images.
> If you are creating a dataset with the circuitboard example images, you have two options:
>
> 1. Create the manifest file using code. The Amazon Lookout for Vision Lab Python Notebook shows how to create the manifest file for the circuitboard example images. Alternatively, use the Datasets example code in the AWS Code Examples Repository.

2. If you've already used the Amazon Lookout for Vision console to create a dataset with the circuitboard example images, reuse the manifest files created for you by Amazon Lookout for Vision. The training and test manifest file locations are `s3://bucket/datasets/project name/train or test/manifests/output/output.manifest`.

If you don't specify `DatasetSource`, an empty dataset is created. In this case, the call to `CreateDataset` is synchronous. Later, you can labeled images to the dataset by calling UpdateDatasetEntries. For example code, see Adding more images (SDK).

If you want to replace a dataset, first delete the existing dataset with DeleteDataset and then create a new dataset of the same dataset type by calling `CreateDataset`. For more information, see Deleting a dataset.

After you create the datasets, you can create the model. For more information, see Training a model (SDK).

You can view the labeled images (JSON lines) within a dataset by calling ListDatasetEntries. You can add labeled images by calling `UpdateDatasetEntries`.

To view information about the test and training datasets, see Viewing your datasets.

To create a dataset (SDK)

1. If you haven't already done so, install and configure the AWS CLI and the AWS SDKs. For more information, see Step 4: Set up the AWS CLI and AWS SDKs.
2. Use the following example code to create a dataset.

   CLI

   Change the following values:

   - `project-name` to the name of the project that you want to associate the dataset with.
   - `dataset-type` to the type of dataset that you want to create (`train` or `test`).
   - `dataset-source` to the Amazon S3 location of the manifest file.
   - `Bucket` to the name of the Amazon S3 bucket that contains the manifest file.
   - `Key` to the path and file name of the manifest file in the Amazon S3 bucket.

```
aws lookoutvision create-dataset --project-name project\
  --dataset-type train or test\
  --dataset-source '{ "GroundTruthManifest": { "S3Object": { "Bucket": "bucket",
 "Key": "manifest file" } } }' \
  --profile lookoutvision-access
```

Python

This code is taken from the AWS Documentation SDK examples GitHub repository. See the full example here.

```
    @staticmethod
    def create_dataset(lookoutvision_client, project_name, manifest_file,
 dataset_type):
        """
        Creates a new Lookout for Vision dataset

        :param lookoutvision_client: A Lookout for Vision Boto3 client.
        :param project_name: The name of the project in which you want to
                             create a dataset.
        :param bucket: The bucket that contains the manifest file.
        :param manifest_file: The path and name of the manifest file.
        :param dataset_type: The type of the dataset (train or test).
        """
        try:
            bucket, key = manifest_file.replace("s3://", "").split("/", 1)
            logger.info("Creating %s dataset type...", dataset_type)
            dataset = {
                "GroundTruthManifest": {"S3Object": {"Bucket": bucket, "Key":
 key}}
            }
            response = lookoutvision_client.create_dataset(
                ProjectName=project_name,
                DatasetType=dataset_type,
                DatasetSource=dataset,
            )
            logger.info("Dataset Status: %s", response["DatasetMetadata"]
["Status"])
            logger.info(
                "Dataset Status Message: %s",
                response["DatasetMetadata"]["StatusMessage"],
```

```
            )
            logger.info("Dataset Type: %s", response["DatasetMetadata"]
["DatasetType"])

            # Wait until either created or failed.
            finished = False
            status = ""
            dataset_description = {}
            while finished is False:
                dataset_description = lookoutvision_client.describe_dataset(
                    ProjectName=project_name, DatasetType=dataset_type
                )
                status = dataset_description["DatasetDescription"]["Status"]

                if status == "CREATE_IN_PROGRESS":
                    logger.info("Dataset creation in progress...")
                    time.sleep(2)
                elif status == "CREATE_COMPLETE":
                    logger.info("Dataset created.")
                    finished = True
                else:
                    logger.info(
                        "Dataset creation failed: %s",
                        dataset_description["DatasetDescription"]
["StatusMessage"],
                    )
                    finished = True

            if status != "CREATE_COMPLETE":
                message = dataset_description["DatasetDescription"]
["StatusMessage"]
                logger.exception("Couldn't create dataset: %s", message)
                raise Exception(f"Couldn't create dataset: {message}")

        except ClientError:
            logger.exception("Service error: Couldn't create dataset.")
            raise
```

Java V2

This code is taken from the AWS Documentation SDK examples GitHub repository. See the full example here.

```
/**
 * Creates an Amazon Lookout for Vision dataset from a manifest file.
 * Returns after Lookout for Vision creates the dataset.
 *
 * @param lfvClient    An Amazon Lookout for Vision client.
 * @param projectName  The name of the project in which you want to create a
 *                     dataset.
 * @param datasetType  The type of dataset that you want to create (train or
 *                     test).
 * @param bucket       The S3 bucket that contains the manifest file.
 * @param manifestFile The name and location of the manifest file within the S3
 *                     bucket.
 * @return DatasetDescription The description of the created dataset.
 */
public static DatasetDescription createDataset(LookoutVisionClient lfvClient,
                String projectName,
                String datasetType,
                String bucket,
                String manifestFile)
                throws LookoutVisionException, InterruptedException {

        logger.log(Level.INFO, "Creating {0} dataset for project {1}",
                        new Object[] { projectName, datasetType });

        // Build the request. If no bucket supplied, setup for empty dataset
 creation.
        CreateDatasetRequest createDatasetRequest = null;

        if (bucket != null && manifestFile != null) {

                InputS3Object s3Object = InputS3Object.builder()
                                .bucket(bucket)
                                .key(manifestFile)
                                .build();

                DatasetGroundTruthManifest groundTruthManifest =
 DatasetGroundTruthManifest.builder()
                                .s3Object(s3Object)
                                .build();

                DatasetSource datasetSource = DatasetSource.builder()
                                .groundTruthManifest(groundTruthManifest)
                                .build();
```

```
                createDatasetRequest = CreateDatasetRequest.builder()
                                .projectName(projectName)
                                .datasetType(datasetType)
                                .datasetSource(datasetSource)
                                .build();
        } else {
                createDatasetRequest = CreateDatasetRequest.builder()
                                .projectName(projectName)
                                .datasetType(datasetType)
                                .build();
        }

        lfvClient.createDataset(createDatasetRequest);

        DatasetDescription datasetDescription = null;

        boolean finished = false;

        // Wait until dataset is created, or failure occurs.
        while (!finished) {

                datasetDescription = describeDataset(lfvClient, projectName,
 datasetType);

                switch (datasetDescription.status()) {
                        case CREATE_COMPLETE:
                                logger.log(Level.INFO, "{0}dataset created for
 project {1}",
                                                new Object[] { datasetType,
 projectName });
                                finished = true;
                                break;
                        case CREATE_IN_PROGRESS:
                                logger.log(Level.INFO, "{0} dataset creating for
 project {1}",
                                                new Object[] { datasetType,
 projectName });

                                TimeUnit.SECONDS.sleep(5);

                                break;

                        case CREATE_FAILED:
```

```
                                logger.log(Level.SEVERE,
                                                "{0} dataset creation failed for
 project {1}. Error {2}",
                                                new Object[] { datasetType,
 projectName,

 datasetDescription.statusAsString() });
                                finished = true;
                                break;
                        default:
                                logger.log(Level.SEVERE, "{0} error when
 creating {1} dataset for project {2}",
                                                new Object[] { datasetType,
 projectName,

 datasetDescription.statusAsString() });
                                finished = true;
                                break;

                }
        }

        logger.log(Level.INFO, "Dataset info. Status: {0}\n Message: {1} }",
                        new Object[] { datasetDescription.statusAsString(),
                                        datasetDescription.statusMessage() });

        return datasetDescription;

}
```

3. Train your model by following the steps at Training a model (SDK).

Labeling images

You can use the Amazon Lookout for Vision console to add or modify the labels assigned to images in your dataset. If you're using the SDK, the labels are part of the manifest file you supply to CreateDataset. You can update the labels for an image by calling UpdateDatasetEntries. For example code, see Adding more images (SDK).

Choosing the model type

The labels you assign to images determine the type of model that Lookout for Vision creates. If your project has a separate test dataset, label the images in the same way.

Image classification model

To create an image classification model, you classify each image as normal or anomalous. For each image, do Classifying images (console).

Image segmentation model

To create an image segmentation model, you classify each image as normal or anomalous. For each anomalous image, you also specify a pixel mask for each anomalous area on the image and an anomaly label for the type of anomaly within the pixel mask. For example, the blue mask in the following image marks the location of a scratch anomaly type on a car. You can specify more than one type of anomaly label in an image. For each image, do Segmenting images (console).

Classifying images (console)

You use the Lookout for Vision console to classify images in a dataset as **normal** or an **anomaly**. Unclasified images aren't used to train your model.

If you're creating an image segmentation model, skip this procedure and do Segmenting images (console), which includes steps to classify images.

> **Note**
>
> If you've just completed Creating your dataset, the console should currently show your model dashboard and you don't need to do steps 1 - 4.

To classify your images (console)

1. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
2. In the left navigation pane, choose **Projects**.
3. In the **Projects** page, choose the project that you want to use.
4. In the left navigation pane of your project, choose **Dataset**.
5. If you have separate training and test datasets, choose the tab for the dataset that you want to use.
6. Choose **Start labeling**.
7. Choose **Select all images on this page**.
8. If the images are normal, choose **Classify as normal**, otherwise choose **Classify as anomaly**. A label appears underneath each picture.
9. If you need to change the label for an image, do the following:

   a. Choose **Anomaly** or **Normal** under the image.

   b. If you can't determine the correct label for an image, magnify the image by choosing the image in the gallery.

   > **Note**
   >
   > You can filter image labels by choosing the desired label, or label state, in the **Filters** section.

10. Repeat steps 7-9 on each page as necessary until all the images in the dataset have been labeled correctly.
11. Choose **Save changes**.
12. If you've finished labeling your images, you can train your model.

Segmenting images (console)

If you're creating an image segmentation model, you must classify images as normal or anomaly. You must also add segmentation information to anomalous images. To specify segmentation information, you first specify anomaly labels for each type of anomaly, such as a dent or scratch,

that you want your model to find. Then you specify an anomaly mask and anomaly label for each anomaly on anomalous images in your dataset.

> **Note**
>
> If you're creating an image classification model, you don't need to segment images and you don't need to specify anomaly labels.

Topics

- Specifying anomaly labels
- Labeling an image
- Segmenting an image with the annotation tool

Specifying anomaly labels

You define an anomaly label for each type of anomaly that's in the dataset images.

Specify anomaly labels

1. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
2. In the left navigation pane, choose **Projects**.
3. In the **Projects** page, choose the project that you want to use.
4. In the left navigation pane of your project, choose **Dataset**.
5. In **Anomaly labels** choose **Add anomaly labels**. If you've previously added an anomaly label, choose **Manage**.
6. In the dialog box, do the following:
   a. Enter the anomaly label that you want to add and choose **Add anomaly label**.
   b. Repeat the previous step until you have entered every anomaly label that you want your model to find.
   c. (Optional) Choose the edit icon to change the label name.
   d. (Optional) Choose the delete icon to delete a new anomaly label. You can't delete anomaly types that your dataset is currently using using.
7. Choose **Confirm** to add the new anomaly labels to the dataset.

After you specify the anomaly labels, label the images by doing Labeling an image.

Labeling an image

To label an image for image segmentation, classify the image as normal or an anomaly. Then, use the annotation tool to segment the image by drawing masks that tightly cover the areas for each type of anomaly present in the image.

To label an image

1. If you have separate training and test datasets, choose the tab for the dataset that you want to use.
2. If you haven't already, specify the anomaly types for your dataset by doing Specifying anomaly labels.
3. Choose **Start labeling**.
4. Choose **Select all images on this page**.
5. If the images are normal, choose **Classify as normal**, otherwise choose **Classify as anomaly**.
6. To change the label for a single image, choose **Normal** or **Anomaly** under the image.

   > **Note**
   >
   > You can filter image labels by choosing the desired label, or label state, in the **Filters** section. You can sort by confidence score in the **Sorting options** section.

7. For each anomalous image, choose the image to open the annotation tool. Add segmentation information by doing Segmenting an image with the annotation tool.
8. Choose **Save changes**.
9. If you've finished labeling your images, you can train your model.

Segmenting an image with the annotation tool

You use the annotation tool to segment an image by marking anomalous areas with a mask.

To segment an image with the annotation tool

1. Open the annotation tool by selecting the image in the dataset gallery. If necessary, choose **Start labeling** to enter labeling mode.

2. In the **Anomaly labels** section choose the anomaly label that you want to mark. If necessary, choose **Add anomaly labels** to add a new anomaly label.
3. Choose a drawing tool at the bottom of the page and draw masks that tightly covers anomalous areas for the anomaly label. The following image is an example of a mask that tightly covers an anomaly.

The following is an example of a poor mask that doesn't tightly cover an anomaly.

4. If you have more images to segment, choose **Next** and repeat steps 2 and 3.
5. Choose **Submit and close** to finish segmenting images.

Training your model

After you have created your datasets and labeled the images, you can train your model. As part of the training process, a test dataset is used. If you have a single dataset project, the images in the dataset are automatically split into a test dataset and a training dataset as part of the training process. If your project has a training and a test dataset, they are used to separately train and test the dataset.

After training is complete, you can evaluate the performance of the model and make any necessary improvements. For more information, see Improving your Amazon Lookout for Vision model.

To train your model, Amazon Lookout for Vision makes a copy of your source training and test images. By default the copied images are encrypted with a key that AWS owns and manages. You can also choose to use your own AWS Key Management Service (KMS) key. For more information, see AWS Key Management Service concepts. Your source images are unaffected.

You can assign metadata to your model in the form of tags. For more information, see Tagging models.

Each time you train a model, a new version of the model is created. If you no longer need a version of a model, you can delete it. For more information, see Deleting a model.

You are charged for the amount of time it takes to successfully train your model. For more information, see Training Hours.

To view the existing models in a project, Viewing your models.

> **Note**
>
> If you've just completed Creating your dataset or Adding images to your dataset. The console should currently show your model dashboard and you don't need to do steps 1 - 4.

Topics

- Training a model (console)
- Training a model (SDK)

Training a model (console)

The following procedure shows you how to train your model using the console.

To train your model (console)

1. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
2. In the left navigation pane, choose **Projects**.
3. In the **Projects** page, choose the project that contains the model that you want to train.
4. On the project details page, choose **Train model**. The **Train model** button is available if you have enough labeled images to train the model. If the button isn't available, add more images until you have enough labeled images.
5. (Optional) If you want to use your own AWS KMS encryption key, do the following:
   a. In **Image data encryption** choose **Customize encryption settings (advanced)**.
   b. In **encryption.aws_kms_key** enter the Amazon Resource Name (ARN) of your key, or choose an existing AWS KMS key. To create a new key, choose **Create an AWS IMS key**.

6. (Optional) if you want to add tags to your model do the following:
   a. In the **Tags** section, choose **Add new tag**.
   b. Enter the following:
      i. The name of the key in **Key**.
      ii. The value of the key in **Value**.
   c. To add more tags, repeat steps 6a and 6b.
   d. (Optional) If you want to remove a tag, choose **Remove** next to the tag that you want to remove. If you are removing a previously saved tag, it is removed when you save your changes.
7. Choose **Train model**.
8. In the **Do you want to train your model?** dialog box, choose **Train model**.
9. In the **Models** view, you can see that training has started and you can check the current status by viewing the `Status` column for the model version. Training a model takes a while to complete.
10. When training is finished, you can evaluate its performance. For more information, see Improving your Amazon Lookout for Vision model.

Training a model (SDK)

You use the CreateModel operation to start the training, testing and evaluation of a model. Amazon Lookout for Vision trains the model using the training and test dataset associated with the project. For more information, see Creating a project (SDK).

Each time you call `CreateModel`, a new version of the model is created. The response from `CreateModel` includes the version of the model.

You are charged for each successful model training. Use the `ClientToken` input parameter to help prevent charges due to unnecessary or accidental repeats of model training by your users. `ClientToken` is an idempotent input parameter that ensures `CreateModel` only completes once for a specific set of parameters — A repeat call to `CreateModel` with the same `ClientToken` value ensures that training isn't repeated. If you don't supply a value for `ClientToken`, the AWS SDK you are using inserts a value for you. This prevents retries after a network error from starting multiple training jobs, but you'll need to provide your own value for your own use cases. For more information, see CreateModel.

Training takes a while to complete. To check the current status, call `DescribeModel` and pass the project name (specified in the call to `CreateProject`) and the model version. The `status` field indicates the current status of the model training. For example code, see Viewing your models (SDK).

If training is successful, you can evaluate model. For more information, see Improving your Amazon Lookout for Vision model.

To view the models that you have created in a project, call `ListModels`. For example code, see Viewing your models.

To train a model (SDK)

1. If you haven't already done so, install and configure the AWS CLI and the AWS SDKs. For more information, see Step 4: Set up the AWS CLI and AWS SDKs.
2. Use the following example code to train a model.

   CLI

   Change the following values:

   - `project-name` to the name of the project that contains the model that you want to create.
   - `output-config` to the location where you want to save training results. Replace the following values:
     - `output bucket` with the name of the Amazon S3 bucket where Amazon Lookout for Vision saves the training results.
     - `output folder` with the name of the folder where you want to save the training results.
     - `Key` with the name of a tag key.
     - `Value` with a value to associate with `tag_key`.

```
aws lookoutvision create-model --project-name "project name"\
  --output-config '{ "S3Location": { "Bucket": "output bucket", "Prefix":
 "output folder" } }'\
  --tags '[{"Key":"Key","Value":"Value"}]' \
  --profile lookoutvision-access
```

Python

This code is taken from the AWS Documentation SDK examples GitHub repository. See the full example here.

```
    @staticmethod
    def create_model(
        lookoutvision_client,
        project_name,
        training_results,
        tag_key=None,
        tag_key_value=None,
    ):
        """
        Creates a version of a Lookout for Vision model.

        :param lookoutvision_client: A Boto3 Lookout for Vision client.
        :param project_name: The name of the project in which you want to create
a
                             model.
        :param training_results: The Amazon S3 location where training results
are stored.
        :param tag_key: The key for a tag to add to the model.
        :param tag_key_value - A value associated with the tag_key.
        return: The model status and version.
        """
        try:
            logger.info("Training model...")
            output_bucket, output_folder = training_results.replace("s3://",
"").split(
                "/", 1
            )
            output_config = {
                "S3Location": {"Bucket": output_bucket, "Prefix": output_folder}
            }
            tags = []
            if tag_key is not None:
                tags = [{"Key": tag_key, "Value": tag_key_value}]

            response = lookoutvision_client.create_model(
                ProjectName=project_name, OutputConfig=output_config, Tags=tags
            )
```

```
            logger.info("ARN: %s", response["ModelMetadata"]["ModelArn"])
            logger.info("Version: %s", response["ModelMetadata"]
["ModelVersion"])
            logger.info("Started training...")

            print("Training started. Training might take several hours to
 complete.")

            # Wait until training completes.
            finished = False
            status = "UNKNOWN"
            while finished is False:
                model_description = lookoutvision_client.describe_model(
                    ProjectName=project_name,
                    ModelVersion=response["ModelMetadata"]["ModelVersion"],
                )
                status = model_description["ModelDescription"]["Status"]

                if status == "TRAINING":
                    logger.info("Model training in progress...")
                    time.sleep(600)
                    continue

                if status == "TRAINED":
                    logger.info("Model was successfully trained.")
                else:
                    logger.info(
                        "Model training failed: %s ",
                        model_description["ModelDescription"]["StatusMessage"],
                    )
                finished = True
        except ClientError:
            logger.exception("Couldn't train model.")
            raise
        else:
            return status, response["ModelMetadata"]["ModelVersion"]
```

Java V2

This code is taken from the AWS Documentation SDK examples GitHub repository. See the full example here.

```
/**
 * Creates an Amazon Lookout for Vision model. The function returns after model
 * training completes. Model training can take multiple hours to complete.
 * You are charged for the amount of time it takes to successfully train a
 model.
 * Returns after Lookout for Vision creates the dataset.
 *
 * @param lfvClient   An Amazon Lookout for Vision client.
 * @param projectName The name of the project in which you want to create a
 *                    model.
 * @param description A description for the model.
 * @param bucket      The S3 bucket in which Lookout for Vision stores the
 *                    training results.
 * @param folder      The location of the training results within the S3
 *                    bucket.
 * @return ModelDescription The description of the created model.
 */
public static ModelDescription createModel(LookoutVisionClient lfvClient, String
 projectName,
                String description, String bucket, String folder)
                throws LookoutVisionException, InterruptedException {

        logger.log(Level.INFO, "Creating model for project: {0}.", new Object[]
 { projectName });

        // Setup input parameters.
        S3Location s3Location = S3Location.builder()
                        .bucket(bucket)
                        .prefix(folder)
                        .build();

        OutputConfig config = OutputConfig.builder()
                        .s3Location(s3Location)
                        .build();

        CreateModelRequest createModelRequest = CreateModelRequest.builder()
                        .projectName(projectName)
                        .description(description)
                        .outputConfig(config)
                        .build();

        // Create and train the model.
```

```
        CreateModelResponse response =
lfvClient.createModel(createModelRequest);

        String modelVersion = response.modelMetadata().modelVersion();
        boolean finished = false;
        DescribeModelResponse descriptionResponse = null;

        // Wait until training finishes or fails.

        do {
                DescribeModelRequest describeModelRequest =
DescribeModelRequest.builder()
                                .projectName(projectName)
                                .modelVersion(modelVersion)
                                .build();

                descriptionResponse =
lfvClient.describeModel(describeModelRequest);

                switch (descriptionResponse.modelDescription().status()) {
                        case TRAINED:
                                logger.log(Level.INFO, "Model training completed
for project {0} version {1}.",
                                                new Object[] { projectName,
modelVersion });
                                finished = true;
                                break;

                        case TRAINING:
                                logger.log(Level.INFO,
                                                "Model training in progress for
project {0} version {1}.",
                                                new Object[] { projectName,
modelVersion });
                                TimeUnit.SECONDS.sleep(60);

                                break;

                        case TRAINING_FAILED:
                                logger.log(Level.SEVERE,
                                                "Model training failed for for
project {0} version {1}.",
                                                new Object[] { projectName,
modelVersion });
```

```
                                finished = true;
                                break;

                        default:
                                logger.log(Level.SEVERE,
                                                "Unexpected error when training
 model project {0} version {1}: {2}.",
                                                new Object[] { projectName,
 modelVersion,

 descriptionResponse.modelDescription()

 .status() });
                                finished = true;
                                break;

                }
        } while (!finished);

        return descriptionResponse.modelDescription();

}
```

3. When training is finished, you can evaluate its performance. For more information, see Improving your Amazon Lookout for Vision model.

Troubleshooting model training

Issues with your manifest file or training images can cause model training to fail. Before retraining your model, check the following potential issues.

Anomaly label colors don't match color of anomalies in mask image

If you are training an image segmentation model, the color of the anomaly label in the manifest file must match the color that's in the mask image. The JSON line for an image in the manifest file has metadata (`internal-color-map`) that tells Amazon Lookout for Vision which color corresponds to an anomaly label. For example, the color for the `scratch` anomaly label in the following JSON line is `#2ca02c`.

```
{
```

```
    "source-ref": "s3://path-to-image",
    "anomaly-label": 1,
    "anomaly-label-metadata": {
        "class-name": "anomaly",
        "creation-date": "2021-10-12T14:16:45.668",
        "human-annotated": "yes",
        "job-name": "labeling-job/classification-job",
        "type": "groundtruth/image-classification",
        "confidence": 1
    },
    "anomaly-mask-ref": "s3://path-to-image",
    "anomaly-mask-ref-metadata": {
        "internal-color-map": {
            "0": {
                "class-name": "BACKGROUND",
                "hex-color": "#ffffff",
                "confidence": 0.0
            },
            "1": {
                "class-name": "scratch",
                "hex-color": "#2ca02c",
                "confidence": 0.0
            },
            "2": {
                "class-name": "dent",
                "hex-color": "#1f77b4",
                "confidence": 0.0
            }
        },
        "type": "groundtruth/semantic-segmentation",
        "human-annotated": "yes",
        "creation-date": "2021-11-23T20:31:57.758889",
        "job-name": "labeling-job/segmentation-job"
    }
}
```

If the colors in the mask image don't match the values in `hex-color`, training fails and you need to update the manifest file.

To update the color values in a manifest file

1. Using a text editor, open the manifest file that you used to create the dataset.

2. For each JSON line (image), check that the colors (`hex-color`) within the `internal-color-map` field match the colors for the anomaly labels in the mask image.

   You can get location of the mask image from *`anomaly-mask`*`-ref` field. Download the image to your computer and use the following code to get the colors in an image.

```
from PIL import Image
img = Image.open('path to local copy of mask file')
colors = img.convert('RGB').getcolors() #this converts the mode to RGB
for color in colors:
    print('#%02x%02x%02x' % color[1])
```

3. For each image with an incorrect color assignment, update the `hex-color` field in the JSON line for the image.
4. Save the update manifest file.
5. Delete the existing dataset from the project.
6. Create a new dataset in the project with the updated manifest file.
7. Train the model.

Alternatively, for steps 5 and 6, you can update individual images in the dataset by calling the UpdateDatasetEntries operation and supplying updated JSON lines for the images you want to update. For example code, see Adding more images (SDK).

Mask images aren't in PNG format

If you are training an image segmentation model, the mask images must be in PNG format. If you create a dataset from a manifest file, make sure the mask images you reference in the *`anomaly-mask`*`-ref` are PNG format. If the mask images are not in PNG format, you need to convert them to PNG format. It is not sufficient to rename the extension for an image file to `.`png.

Mask images that you create in the Amazon Lookout for Vision console or with a SageMaker AI Ground Truth job are created in PNG format. You do not need to change the format of these images.

To correct non-PNG format mask images in a manifest file

1. Using a text editor, open the manifest file that you used to create the dataset.

2. For each JSON lines (image) make image sure that the `anomaly-mask-ref` references a PNG format image. For more information, see Creating a manifest file.
3. Save the updated manifest file.
4. Delete the existing dataset from the project.
5. Create a new dataset in the project with the updated manifest file.
6. Train the model.

Segmentation or classification labels are inaccurate or missing

Missing or inaccurate labels can cause training to fail or create a model that performs poorly. We recommend that you label all images in your dataset. If you don't label all images and model training fails, or your model performs poorly, add more images.

Check the following:

- If you are creating a segmentation model, masks must tightly cover the anomalies on your dataset images. To check the masks in your dataset, view the images in the project's dataset gallery. If necessary, redraw the image masks. For more information, see Segmenting images (console).

- Make sure that anomalous images in your dataset images are classified. If you're creating an image segmentation model, make sure anomalous images have anomaly labels and image masks.

  It's important to remember which type of model (segmentation or classification) you are creating. A classification model doesn't require image masks on anomalous images. Don't add masks to dataset images intended for a classification model.

  To update missing labels

  1. Open the project's dataset gallery.
  2. Filter **Unlabeled** images to see which images do not have labels.
  3. Do one of the following:
     - If you are creating an image classification model, classify each unlabeled image.

- If you are creating an image segmentation model, classify and segment each unlabeled image.

4. If you are creating an image segmentation model, add masks to any classified anomalous images that are missing masks.
5. Train the model.

If you choose not to fix poor or missing labels, we recommend that you add more labeled images or remove the affected images from the dataset. You can add more from the console or by using the UpdateDatasetEntries operation. For more information, see Adding images to your dataset.

If you choose to remove the images, you must recreate the dataset without the affected images, as you can't delete an image from a dataset. For more information, see Removing images from your dataset.

Improving your Amazon Lookout for Vision model

During training Lookout for Vision tests your model with the test dataset and uses the results to create performance metrics. You can use performance metrics to evaluate the performance of your model. If necessary, you can take steps to improve your datasets and then retrain your model.

If you're satisfied with the performance of your model, you can begin to use it. For more information, see Running your trained Amazon Lookout for Vision model.

Topics

- Step 1: Evaluate the performance of your model
- Step 2: Improve your model
- Viewing performance metrics
- Verifying your model with a trial detection task

Step 1: Evaluate the performance of your model

You can access the performance metrics from the console and from the DescribeModel operation. Amazon Lookout for Vision provides summary performance metrics for the test dataset and the predicted results for all individual images. If your model is a segmentation model, the console also shows summary metrics for each anomaly label.

To view the performance metrics and test image predictions in the console, see Viewing performance metrics (console). For information about accessing performance metrics and test image predictions with the `DescribeModel` operation, see Viewing performance metrics (SDK).

Image classification metrics

Amazon Lookout for Vision provides the following summary metrics for the classifications that a model makes during testing:

- Precision
- Recall
- F1 score

Image segmentation model metrics

If the model is an image segmentation model, Amazon Lookout for Vision provides summary image classification metrics and summary performance metrics for each anomaly label:

- F1 score
- Average Intersection over Union (IoU)

Precision

The precision metric answers the question – *When the model predicts that an image contains an anomaly, how often is that prediction correct?*

Precision is a useful metric for situations where the cost of a false positive is high. For example, the cost of removing a machine part that is not defective from an assembled machine.

Amazon Lookout for Vision provides a summary precision metric value for the entire test dataset.

Precision is the fraction of correctly predicted anomalies (true positives) over all predicted anomalies (true and false positives). The formula for precision is as follows.

Precision value = true positives / (true positives + false positives)

The possible values for precision range from 0–1. The Amazon Lookout for Vision console displays precision as a percentage value (0–100).

A higher precision value indicates that more of the predicted anomalies are correct. For example, suppose your model predicts that 100 images are anomalous. If 85 of the predictions are correct (the true positives) and 15 are incorrect (the false positives), the precision is calculated as follows:

85 *true positives* / (85 *true positives* + 15 *false positives*) = **0.85** precision value

However, if the model only predicts 40 images correctly out of 100 anomaly predictions, the resulting precision value is lower at **0.40** (that is, 40 / (40 + 60) = 0.40). In this case, your model is making more incorrect predictions than correct predictions. To fix this, consider making improvements to your model. For more information, see Step 2: Improve your model.

For more information, see Precision and recall.

Recall

The recall metric answers the question - *Of the total number of anomalous images in the test dataset, how many are correctly predicted as anomalous?*

The recall metric is useful for situations where the cost of a false negative is high. For example, when the cost of not removing a defective part is high. Amazon Lookout for Vision provides a summary recall metric value for the entire test dataset.

Recall is the fraction of the anomalous test images that were detected correctly. It is a measure of how often the model can correctly predict an anomalous image, when it's actually present in the images of your test dataset. The formula for recall is calculated as follows:

Recall value = true positives / (true positives + false negatives)

The range for recall is 0–1. The Amazon Lookout for Vision console displays recall as a percentage value (0–100).

A higher recall value indicates that more of the anomalous images are correctly identified. For example, suppose the test dataset contains 100 anomalous images. If the model correctly detects 90 of the 100 anomalous images, then the recall is as follows:

90 *true positives* / (90 *true positives* + 10 *false negatives*) = **0.90** recall value

A recall value of 0.90 indicates that your model is correctly predicting most of the anomalous images in the test dataset. If the model only predicts 20 of the anomalous images correctly, the recall is lower at **0.20** (that is, 20 / (20 + 80) = 0.20).

In this case, you should consider making improvements to your model. For more information, see Step 2: Improve your model.

For more information, see Precision and recall.

F1 score

Amazon Lookout for Vision provides an average model performance score for the test dataset. Specifically, model performance for anomaly classification is measured by the F1 score metric, which is the harmonic mean of the precision and recall scores.

F1 score is an aggregate measure that takes into account both precision and recall. The model performance score is a value between 0 and 1. The higher the value, the better the model is

performing for both recall and precision. For example, for a model with precision of 0.9 and a recall of 1.0, the F1 score is 0.947.

If the model isn't performing well, for example, with a low precision of 0.30 and a high recall of 1.0, the F1 score is 0.46. Similarly, if the precision is high (0.95) and the recall is low (0.20), the F1 score is 0.33. In both cases, the F1 score is low, which indicates problems with the model.

For more information, see F1 score.

Average Intersection over Union (IoU)

The average percentage overlap between the anomaly masks in the test images and the anomaly masks that the model predicts for the test images. Amazon Lookout for Vision returns the Average IoU for each anomaly label and is only returned by image segmentation models.

A low percentage value indicates that the model isn't accurately matching its predicted masks for a label with the masks in the test images.

The following image has a low IoU. The orange mask is the prediction from the model and doesn't tightly cover the blue mask that represents the mask in a test image.

The following image has a higher IoU. The blue mask (test image) is tightly covered by the orange mask (predicted mask).

Testing results

During testing, the model predicts classification for each test image in the test dataset. The result for each prediction is compared to the label (normal or anomaly) of the corresponding test image as follows:

- Correctly predicting that an image is anomalous is considered a *true positive*.
- Incorrectly predicting that an image is anomalous is considered a *false positive*.
- Correctly predicting that an image is normal is considered a *true negative*.
- Incorrectly predicting that an image is normal is considered a *false negative*.

If the model is a segmentation model, the model also predicts masks and anomaly labels for the location of anomalies on the test image.

Amazon Lookout for Vision uses the results of the comparisons to generate the performance metrics.

Step 2: Improve your model

The performance metrics might show that you can improve your model. For example, if the model doesn't detect all anomalies in the test dataset, your model has low recall (that is, the recall metric has a low value). If you need to improve your model, consider the following:

- Check that the training and test dataset images are properly labeled.
- Reduce the variability of image capture conditions such as lighting and object pose, and train your model on objects of the same type.
- Ensure that your images show only the required content. For example, if your project detects anomalies in machine parts, make sure that no other objects are in your images.

- Add more labeled images to your train and test datasets. If your test dataset has excellent recall and precision but the model performs poorly when deployed, your test dataset might not be representative enough and you need to extend it.
- If your test dataset results in poor recall and precision, consider how well the anomalies and image capture conditions in the training and test datasets match. If your training images aren't representative of the expected anomalies and conditions, but images in the test images are, add images to the training training dataset with the expected anomalies and conditions. If the test dataset images aren't in the expected conditions, but the training images are, update the test dataset.

  For more information, see Adding more images. An alternative way to add labeled images to your training dataset is to run a trial detection task and verify the results. You can then add the verified images to the training dataset. For more information, see Verifying your model with a trial detection task.
- Ensure you have sufficiently diverse normal and anomalous images in your training and test dataset. The images must represent the type of normal and anomalous images that your model will encounter. For example, when analyzing circuit boards, your normal images should represent the variations in position and soldering of components, such as resistors and transistors. The anomalous images should represent the different types of anomalies that the system might encounter, such as misplaced or missing components.
- If your model has low Average IoU for detected anomaly types, check the mask outputs from the segmentation model. For some use cases, such as scratches, the model might output scratches that are very close to groundtruth scratches in the test images, but with a low pixel overlap. For example, two parallel lines that are 1 pixel distance apart. In those cases, Average IOU is an unreliable indicator to measure prediction success.
- If the image size is small, or the image resolution is low, consider capturing images at a higher resolution. Image dimensions can range from 64 x 64 pixels up to 4096 pixels X 4096 pixels.
- If the anomaly size is small, consider dividing the images into separate tiles and use the tiled images for training and testing. This lets the model see defects at a larger size in an image.

After you have improved your training and test dataset, retrain and re-evaluate your model. For more information, see Training your model.

If the metrics show that you model has acceptable performance, you can verify its performance by adding the results of a trial detection task to the test dataset. After retraining, the performance

metrics should confirm the performance metrics from the previous training. For more information, see Verifying your model with a trial detection task.

Viewing performance metrics

You can get performance metrics from the console and by calling the `DescribeModel` operation.

Topics

- Viewing performance metrics (console)
- Viewing performance metrics (SDK)

Viewing performance metrics (console)

After training is complete, the console displays the performance metrics.

The Amazon Lookout for Vision console shows the following performance metrics for the classifications made during testing:

- Precision
- Recall
- F1 score

If the model is a segmentation model, the console also shows the following performance metrics for each anomaly label:

- The number of test images where the anomaly label was found.
- F1 score
- Average Intersection over Union (IoU)

The test results overview section shows you the total correct and incorrect predictions for images in the test dataset. You can also see the predicted and actual label assignments for individual images in the test dataset.

The following procedure shows how to get the performance metrics from a project's model list view.

To view performance metrics (console)

1. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
2. Choose **Get started**.
3. In the left navigation pane, choose **Projects**.
4. In the projects view, choose the project that contains the version of the model that you want to view.
5. In the left navigation pane, under the project name, choose **Models**.
6. In the models list view, choose the versions of the model that you want to view.
7. On the model details page, view the performance metrics on the **Performance metrics** tab.
8. Note the following:
   a. The **Model performance metrics** section contains overall model metrics (precision, recall, F1 score) for the classification predictions that the model made for the test images.
   b. If the model is an image segmentation model, the **Performance per label** section contains the number of test images where the anomaly label was found. You also see metrics (F1 score, Average IoU) for each anomaly label.
   c. The **Test results overview** section provides the results for each test image that Lookout for Vision uses to evaluate the model. It includes the following:
      - The total number of correct (true positive) and incorrect (false negative) classification predictions (normal or anomaly) for all test images.
      - The classification prediction for each test image. If you see **Correct** under an image, the predicted classification matches the actual classification for the image. Otherwise the model didn't correctly classify the image.
      - With an image segmentation model, you see anomaly labels that the model assigned to the image and masks on the image that match the colors of the anomaly labels.

Viewing performance metrics (SDK)

You can use the DescribeModel operation to get the summary performance metrics (classification) for the model, the evaluation manifest, and the evaluation results for a model.

Getting the summary performance metrics

The summary performance metrics for the classification predictions made by the model during testing (Precision, Recall, and F1 score) are returned in the `Performance` field returned by a call to `DescribeModel`.

```
        "Performance": {
            "F1Score": 0.8,
            "Recall": 0.8,
            "Precision": 0.9
        },
```

The `Performance` field doesn't include the anomaly label performance metrics returned by a segmentation model. You can get them from the `EvaluationResult` field. For more information, see Reviewing the evaluation result.

For information about summary performance metrics see Step 1: Evaluate the performance of your model. For example code, see Viewing your models (SDK).

Using the evaluation manifest

The evaluation manifest provides test prediction metrics for the individual images used to test a model. For each image in the test dataset, a JSON line contains the original test (ground truth) information and the model's prediction for the image. Amazon Lookout for Vision stores the evaluation manifest in an Amazon S3 bucket. You can get the location from the `EvaluationManifest` field in the response from `DescribeModel` operation.

```
"EvaluationManifest": {
            "Bucket": "lookoutvision-us-east-1-nnnnnnnnnn",
            "Key": "my-sdk-project-model-output/EvaluationManifest-my-sdk-
project-1.json"
        }
```

The file name format is `EvaluationManifest-`*`project name`*`.json`. For example code, see Viewing your models.

In the following sample JSON line, the `class-name` is the ground truth for the contents of the image. In this example the image contains an anomaly. The `confidence` field shows the confidence that Amazon Lookout for Vision has in the prediction.

```
{
    "source-ref"*: "s3://customerbucket/path/to/image.jpg",
    "source-ref-metadata": {
        "creation-date": "2020-05-22T21:33:37.201882"
    },

    // Test dataset ground truth
    "anomaly-label": 1,
    "anomaly-label-metadata": {
        "class-name": "anomaly",
        "type": "groundtruth/image-classification",
        "human-annotated": "yes",
        "creation-date": "2020-05-22T21:33:37.201882",
        "job-name": "labeling-job/anomaly-detection"
    },
    // Anomaly label detected by Lookout for Vision
    "anomaly-label-detected": 1,
    "anomaly-label-detected-metadata": {
        "class-name": "anomaly",
        "confidence": 0.9,
        "type": "groundtruth/image-classification",
        "human-annotated": "no",
        "creation-date": "2020-05-22T21:33:37.201882",
        "job-name": "training-job/anomaly-detection",
        "model-arn": "lookoutvision-some-model-arn",
        "project-name": "lookoutvision-some-project-name",
        "model-version": "lookoutvision-some-model-version"
    }
}
```

Reviewing the evaluation result

The evaluation result has the following aggregate performance metrics (classification) for the entire set of test images:

- Precision
- Recall
- ROC curve (not shown in console)
- Average precision (not shown in console)
- F1 score

The evaluation result also includes the number of images used for training and testing the model.

If the model is a segmentation model, the evaluation result also includes the following metrics for each anomaly label found in the test dataset:

- Precision
- Recall
- F1 score
- Average Intersection over Union (IoU)

Amazon Lookout for Vision stores the evaluation result in an Amazon S3 bucket. You can get the location by checking the value of `EvaluationResult` in the response from `DescribeModel` operation.

```
"EvaluationResult": {
            "Bucket": "lookoutvision-us-east-1-nnnnnnnnnn",
            "Key": "my-sdk-project-model-output/EvaluationResult-my-sdk-project-1.json"
        }
```

The file name format is `EvaluationResult-`*`project name`*`.json`. For an example, see Viewing your models.

The following schema shows the evaluation result.

```
    {
        "Version": 1,
        "EvaluationDetails":
        {
            "ModelArn": "string", // The Amazon Resource Name (ARN) of the model
 version.
            "EvaluationEndTimestamp": "string",   // The UTC date and time that
 evaluation finished.
            "NumberOfTrainingImages": int,        // The number of images that were
 successfully used for training.
            "NumberOfTestingImages": int          // The number of images that were
 successfully used for testing.
        },
        "AggregatedEvaluationResults":
        {
            "Metrics":
```

```
            {                       //Classification metrics.
                "ROCAUC": float,            // ROC area under the curve.
                "AveragePrecision": float,  // The average precision of the model.
                "Precision": float,         // The overall precision of the model.
                "Recall": float,            // The overall recall of the model.
                "F1Score": float,           // The overal F1 score for the model.

                "PixelAnomalyClassMetrics":     //Segmentation metrics.
                [
                    {
                        "Precision": float,       // The precision for the anomaly
label.
                        "Recall": float,          // The recall for the anomaly label.
                        "F1Score": float,         // The F1 score for the anomaly
label.
                        "AIOU" : float,           // The average Intersection Over
Union for the anomaly label.
                        "ClassName": "string"   // The anomaly label.
                    }
                ]
            }
        }
    }
```

Verifying your model with a trial detection task

If you want to verify or improve the quality of your model, you can run a trial detection task. A trial detection task detects anomalies in new images that you supply.

You can verify the detection results and add the verified images to your dataset. If you have separate training and test datasets, the verified images are added to the training dataset.

You can verify images from your local computer or images located in an Amazon S3 bucket. If you want to add verified images to the dataset, images located in an S3 bucket must be in the same S3 bucket as the images in your dataset.

> **Note**
>
> To run a trial detection task, ensure that your S3 bucket has versioning enabled. For more information, see Using versioning. The console bucket is created with versioning enabled.

By default your images are encrypted with a key that AWS owns and manages. You can also choose to use your own AWS Key Management Service (KMS) key. For more information, see AWS Key Management Service concepts.

Topics

- Running a trial detection task
- Verifying trial detection results
- Correcting segmentation labels with the annotation tool

Running a trial detection task

Perform the following steps to run a trial detection task.

To run a trial detection (console)

1. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
2. Choose **Get started**.
3. In the left navigation pane, choose **Projects**.
4. In the projects view, choose the project that contains the version of the model that you want to view.
5. In the left navigation pane, under the project name, choose **Trial detections**.
6. In the trial detections view, choose the **Run trial detection**.
7. On the **Run trial detection** page, enter a name for your trial detection task in **Task name**.
8. In **Choose model**, choose the version of that model that you want to use.
9. Import the images according to the source of the images as follows:

   - If you are importing your source images from an Amazon S3 bucket, enter the **S3 URI**.

     > **Tip**
     >
     > If you're using the Getting Started example images, use the *extra_images* folder. The Amazon S3 URI is `s3://your bucket/circuitboard/extra_images`.

   - If you are uploading images from your computer, add the images after you choose **Detect anomalies**.

10. (Optional) If you want to use your own AWS KMS encryption key, do the following:
    a. For **Image data encryption**, choose **Customize encryption settings (advanced)**.
    b. In **encryption.aws_kms_key**, enter the Amazon Resource Name (ARN) of your key, or choose an existing AWS KMS key. To create a new key, choose **Create an AWS IMS key**.
11. Choose **Detect anomalies** and then choose **Run trial detection** to start the trial detection task.
12. Check the current status in the trial detections view. The trial detection might take a while to complete.

Verifying trial detection results

Verifying the results of a trial detection can help you improve your model.

If the performance metrics are poor, improve your model by running a trial detection and then add verified images to the dataset (training dataset, if you have a separate datasets).

If the model's performance metrics are good, but the results of a trial detection are poor, you can improve your model by adding verified images to the dataset (training dataset). If you have a separate test dataset, consider adding more images to the test dataset.

After you add verified images to your dataset, retrain and re-evaluate your model. For more information, see Training your model.

To verify the results of a trial detection

1. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
2. In the left navigation pane, choose **Projects**.
3. In the **Projects** page, choose the project that you want to use. The dashboard for your project is displayed.
4. In the left navigation pane, choose **Trial detections**.
5. Choose the trial detection that you want to verify.
6. On the trial detection page, choose **Verify machine predictions**.
7. Choose **Select all images on this page**.
8. If the predictions are correct, choose **Verify as correct**. Otherwise, choose **Verify as incorrect**. The prediction and prediction confidence score is shown under each image.

9. If you need to change the label for an image, do the following:

    a. Choose **Correct** or **Incorrect** under the image.

    b. If you can't determine the correct label for an image, magnify the image by choosing the image in the gallery.

> **Note**
>
> You can filter image labels by choosing the desired label, or label state, in the **Filters** section. You can sort by confidence score in the **Sorting options** section.

10. If your model is a segmentation model and the mask or anomaly label for an image is wrong, choose **Anomalous area** under the image and open the annotation tool. Update the segmentation information by doing Correcting segmentation labels with the annotation tool.
11. Repeat steps 7-10 on each page as necessary until all the images have been verified.
12. Choose **Add verified images to dataset**. If you have separate datasets, the images are added to the training dataset.
13. Retrain your model. For more information, see the section called “Training your model”.

Correcting segmentation labels with the annotation tool

You use the annotation tool to segment an image by marking anomalous areas with a mask.

To correct the segmentation labels for an image with the annotation tool

1. Open the annotation tool by selecting **anomalous area** under an image in the dataset gallery.
2. If the anomaly label for a mask isn't correct, choose the mask and then choose the correct anomaly label under **Anomaly labels**. If necessary, choose **Add anomaly label** to add a new anomaly label.
3. If the mask isn't correct, choose a drawing tool at the bottom of the page and draw masks that tightly covers anomalous areas for the anomaly label. The following image is an example of a mask that tightly covers an anomaly.

The following is an example of a poor mask that doesn't tightly cover an anomaly.

4. If you have more images to correct, choose **Next** and repeat steps 2 and 3.
5. Choose **Submit and close** to finish updating images.

Running your trained Amazon Lookout for Vision model

To detect anomalies in images with your model, you must first start your model with the StartModel operation. The Amazon Lookout for Vision console provides AWS CLI commands that you can use to start and stop your model. This section includes example code that you can use.

After your model starts, you can use the `DetectAnomalies` operation to detect anomalies in an image. For more information, see Detecting anomalies in an image.

Topics

- Inference units
- Availability Zones
- Starting your Amazon Lookout for Vision model
- Stopping your Amazon Lookout for Vision model

Inference units

When you start your model, Amazon Lookout for Vision provisions a minimum of one compute resource, known as an inference unit. You specify the number of inference units to use in the `MinInferenceUnits` input parameter to the `StartModel` API. The default allocation for a model is 1 inference unit.

> **Important**
>
> You are charged for the number of hours that your model is running and for the number of inference units that your model uses while it's running, based on how you configure the running of your model. For example, if you start the model with two inference units and use the model for 8 hours, you are charged for 16 inference hours (8 hours running time * two inference units). For more information, see Amazon Lookout for Vision Pricing. If you don't explicitly stop your model by calling StopModel, you are charged even if you are not actively analyzing images with your model.

The transactions per second (TPS) that a single inference unit supports is affected by the following:

- The algorithm that Lookout for Vision uses to train the model. When you train a model, multiple models are trained. Lookout for Vision selects the model with the best performance based on the size of the dataset and its composition of normal and anomalous images.
- Higher resolution images require more time for analysis.
- Smaller sized images (measured in MBs) are analyzed faster than larger images.

Managing throughput with inference units

You can increase or decrease the throughput of your model depending on the demands on your application. To increase throughput, use additional inference units. Each additional inference unit increases your processing speed by one inference unit. For information about calculating the number of inference units that you need, see Calculate inference units for Amazon Rekognition Custom Labels and Amazon Lookout for Vision models. If you want to change the supported throughput of your model, you have two options:

Manually add or remove inference units

Stop the model and then restart with the required number of inference units. The disadvantage with this approach is that the model can't receive requests while it's restarting and can't be used to handle spikes in demand. Use this approach if your model has steady throughput and your use case can tolerate 10–20 minutes of downtime. An example would be if you want to batch calls to your model using a weekly schedule.

Auto-scale inference units

If your model has to accommodate spikes in demand, Amazon Lookout for Vision can automatically scale the number of inference units that your model uses. As demand increases, Amazon Lookout for Vision adds additional inference units to the model and removes them when demand decreases.

To let Lookout for Vision automatically scale inference units for a model, start the model and set the maximum number of inference units that it can use by using the `MaxInferenceUnits` parameter. Setting a maximum number of inference units lets you manage the cost of running the model by limiting the number of inference units available to it. If you don't specify a maximum number of units, Lookout for Vision won't automatically scale your model, only using the number of inference units that you started with. For information regarding the maximum number of inference units, see Service Quotas.

You can also specify a minimum number of inference units by using the `MinInferenceUnits` parameter. This lets you specify the minimum throughput for your model, where a single inference unit represents 1 hour of processing time.

> **Note**
>
> You can't set the maximum number of inference units with the Lookout for Vision console. Instead, specify the `MaxInferenceUnits` input parameter to the `StartModel` operation.

Lookout for Vision provides the following Amazon CloudWatch Logs metrics that you can use to determine the current automatic scaling status for a model.

Metric	Description
`DesiredInferenceUnits`	The number of inference units to which Lookout for Vision is scaling up or down.
`InServiceInferenceUnits`	The number of inference units that the model is using.

If `DesiredInferenceUnits` = `InServiceInferenceUnits`, Lookout for Vision is not currently scaling the number of inference units.

If `DesiredInferenceUnits` > `InServiceInferenceUnits`, Lookout for Vision is scaling up to the value of `DesiredInferenceUnits`.

If `DesiredInferenceUnits` < `InServiceInferenceUnits`, Lookout for Vision is scaling down to the value of `DesiredInferenceUnits`.

For more information regarding the metrics returned by Lookout for Vision and filtering dimensions, see Monitoring Lookout for Vision with Amazon CloudWatch.

To find out the maximum number of inference units that you requested for a model, call DescribeModel and check the `MaxInferenceUnits` field in the response.

Availability Zones

Amazon Lookout for Vision; distributes inference units across multiple Availability Zones within an AWS Region to provide increased availability. For more information, see Availability Zones. To help protect your production models from Availability Zone outages and inference unit failures, start your production models with at least two inference units.

If an Availability Zone outage occurs, all inference units in the Availability Zone are unavailable and model capacity is reduced. Calls to DetectAnomalies are redistributed across the remaining inference units. Such calls succeed if they don't exceed the supported Transactions Per Seconds (TPS) of the remaining inference units. After AWS repairs the Availability Zone, the inference units are restarted, and full capacity is restored.

If a single inference unit fails, Amazon Lookout for Vision automatically starts a new inference unit in the same Availability Zone. Model capacity is reduced until the new inference unit starts.

Starting your Amazon Lookout for Vision model

Before you can use an Amazon Lookout for Vision model to detect anomalies, you must first start the model. You start a model by calling the StartModel API and passing the following:

- **ProjectName** – The name of the project that contains the model that you want to start.
- **ModelVersion** – The version of the model that you want to start.
- **MinInferenceUnits** – The minimum number of inference units. For more information, see Inference units.
- (Optional) **MaxInferenceUnits** – The maximum number of inference units that Amazon Lookout for Vision can use to automatically scale the model. For more information, see Auto-scale inference units.

The Amazon Lookout for Vision console provides example code that you can use to start and stop a model.

> **Note**
>
> You are charged for the amount of the time that your model is running. To stop a running model, see Stopping your Amazon Lookout for Vision model.

You can use the AWS SDK to view running models across all AWS Regions in which Lookout for Vision is available. For example code, see find_running_models.py.

Topics

- Starting your model (console)
- Starting your Amazon Lookout for Vision model (SDK)

Starting your model (console)

The Amazon Lookout for Vision console provides a AWS CLI command that you can use to start a model. After the model starts, you can start detecting anomalies in images. For more information, see Detecting anomalies in an image.

To start a model (console)

1. If you haven't already done so, install and configure the AWS CLI and the AWS SDKs. For more information, see Step 4: Set up the AWS CLI and AWS SDKs.
2. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
3. Choose **Get started**.
4. In the left navigation pane, choose **Projects**.
5. On the **Projects** resources page, choose the project that contains the trained model that you want to start.
6. In the **Models** section, choose the model that you want to start.
7. On the model's details page, choose **Use model** and then choose **Integrate API to the cloud**.

   > **Tip**
   >
   > If you want to deploy your model to an edge device, choose **Create model packaging job**. For more information, see Packaging your Amazon Lookout for Vision model.

8. Under **AWS CLI commands**, copy the AWS CLI command that calls `start-model`.
9. At the command prompt, enter the `start-model` command that you copied in the previous step. If you are using the `lookoutvision` profile to get credentials, add the `--profile lookoutvision-access` parameter.

10. In the console, choose **Models** in the left navigation page.
11. Check the **Status** column for the current status of the model, When the status is **Hosted**, you can use the model to detect anomalies in images. For more information, see Detecting anomalies in an image.

Starting your Amazon Lookout for Vision model (SDK)

You start a model by calling the StartModel operation.

A model might take a while to start. You can check the current status by calling DescribeModel. For more information, see Viewing your models.

To start your model (SDK)

1. If you haven't already done so, install and configure the AWS CLI and the AWS SDKs. For more information, see Step 4: Set up the AWS CLI and AWS SDKs.
2. Use the following example code to start a model.

   CLI

   Change the following values:

   - `project-name` to the name of the project that contains the model that you want to start.
   - `model-version` to the version of the model that you want to start.
   - `--min-inference-units` to the number of inference units that you want to use.
   - (Optional) `--max-inference-units` to the maximum number of inference units that Amazon Lookout for Vision can use to automatically scale the model.

```
aws lookoutvision start-model --project-name "project name"\
  --model-version model version\
  --min-inference-units minimum number of units\
  --max-inference-units max number of units \
  --profile lookoutvision-access
```

Python

This code is taken from the AWS Documentation SDK examples GitHub repository. See the full example here.

```
    @staticmethod
    def start_model(
            lookoutvision_client, project_name, model_version,
min_inference_units, max_inference_units = None):
        """
        Starts the hosting of a Lookout for Vision model.

        :param lookoutvision_client: A Boto3 Lookout for Vision client.
        :param project_name:  The name of the project that contains the version
of the
                              model that you want to start hosting.
        :param model_version: The version of the model that you want to start
hosting.
        :param min_inference_units: The number of inference units to use for
hosting.
        :param max_inference_units: (Optional) The maximum number of inference
units that
        Lookout for Vision can use to automatically scale the model.
        """
        try:
            logger.info(
                "Starting model version %s for project %s", model_version,
project_name)

            if max_inference_units is None:
                lookoutvision_client.start_model(
                    ProjectName = project_name,
                    ModelVersion = model_version,
                    MinInferenceUnits = min_inference_units)

            else:
                lookoutvision_client.start_model(
                    ProjectName = project_name,
                    ModelVersion = model_version,
                    MinInferenceUnits = min_inference_units,
                    MaxInferenceUnits = max_inference_units)

            print("Starting hosting...")
```

```
            status = ""
            finished = False

            # Wait until hosted or failed.
            while finished is False:
                model_description = lookoutvision_client.describe_model(
                    ProjectName=project_name, ModelVersion=model_version)
                status = model_description["ModelDescription"]["Status"]

                if status == "STARTING_HOSTING":
                    logger.info("Host starting in progress...")
                    time.sleep(10)
                    continue

                if status == "HOSTED":
                    logger.info("Model is hosted and ready for use.")
                    finished = True
                    continue

                logger.info("Model hosting failed and the model can't be used.")
                finished = True

            if status != "HOSTED":
                logger.error("Error hosting model: %s", status)
                raise Exception(f"Error hosting model: {status}")
        except ClientError:
            logger.exception("Couldn't host model.")
            raise
```

Java V2

This code is taken from the AWS Documentation SDK examples GitHub repository. See the full example here.

```
/**
 * Starts hosting an Amazon Lookout for Vision model. Returns when the model has
 * started or if hosting fails. You are charged for the amount of time that a
 * model is hosted. To stop hosting a model, use the StopModel operation.
 *
 * @param lfvClient   An Amazon Lookout for Vision client.
 * @param projectName The name of the project that contains the model that you
 *                    want to host.
```

```
 * @modelVersion The version of the model that you want to host.
 * @minInferenceUnits The number of inference units to use for hosting.
 * @maxInferenceUnits The maximum number of inference units that Lookout for
 *                    Vision can use for automatically scaling the model. If the
 *                    value is null, automatic scaling doesn't happen.
 * @return ModelDescription The description of the model, which includes the
 *         model hosting status.
 */
public static ModelDescription startModel(LookoutVisionClient lfvClient, String
 projectName, String modelVersion,
        Integer minInferenceUnits, Integer maxInferenceUnits) throws
 LookoutVisionException, InterruptedException {

    logger.log(Level.INFO, "Starting Model version {0} for project {1}.",
            new Object[] { modelVersion, projectName });

    StartModelRequest startModelRequest = null;

    if (maxInferenceUnits == null) {

        startModelRequest =
 StartModelRequest.builder().projectName(projectName).modelVersion(modelVersion)
                .minInferenceUnits(minInferenceUnits).build();
    } else {
        startModelRequest =
 StartModelRequest.builder().projectName(projectName).modelVersion(modelVersion)

 .minInferenceUnits(minInferenceUnits).maxInferenceUnits(maxInferenceUnits).build();
    }

    // Start hosting the model.
    lfvClient.startModel(startModelRequest);

    DescribeModelRequest describeModelRequest =
 DescribeModelRequest.builder().projectName(projectName)
            .modelVersion(modelVersion).build();

    ModelDescription modelDescription = null;

    boolean finished = false;
    // Wait until model is hosted or failure occurs.
    do {
```

```
        modelDescription =
lfvClient.describeModel(describeModelRequest).modelDescription();

        switch (modelDescription.status()) {

        case HOSTED:
            logger.log(Level.INFO, "Model version {0} for project {1} is
running.",
                    new Object[] { modelVersion, projectName });
            finished = true;
            break;

        case STARTING_HOSTING:
            logger.log(Level.INFO, "Model version {0} for project {1} is
starting.",
                    new Object[] { modelVersion, projectName });

            TimeUnit.SECONDS.sleep(60);

            break;
        case HOSTING_FAILED:
            logger.log(Level.SEVERE, "Hosting failed for model version {0} for
project {1}.",
                    new Object[] { modelVersion, projectName });
            finished = true;
            break;

        default:
            logger.log(Level.SEVERE, "Unexpected error when hosting model
version {0} for project {1}: {2}.",
                    new Object[] { projectName, modelVersion,
modelDescription.status() });
            finished = true;
            break;

        }

    } while (!finished);

    logger.log(Level.INFO, "Finished starting model version {0} for project {1}
status: {2}",
            new Object[] { modelVersion, projectName,
modelDescription.statusMessage() });
```

```
    return modelDescription;

}
```

3. If the output of the code is `Model is hosted and ready for use`, you can use the model to detect anomalies in images. For more information, see Detecting anomalies in an image.

Stopping your Amazon Lookout for Vision model

To stop a running model, you call the `StopModel` operation and pass the following:

- **Project** – The name of the project that contains the model that you want to stop.
- **ModelVersion** – The version of the model that you want to stop.

The Amazon Lookout for Vision console provides example code that you can use to stop a model.

> **Note**
>
> You are charged for the amount of the time that your model is running.

Topics

- Stopping your model (console)
- Stopping your Amazon Lookout for Vision model (SDK)

Stopping your model (console)

Perform the steps in the following procedure to stop your model using the console.

To stop your model (console)

1. If you haven't already done so, install and configure the AWS CLI and the AWS SDKs. For more information, see Step 4: Set up the AWS CLI and AWS SDKs.
2. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
3. Choose **Get started**.

4. In the left navigation pane, choose **Projects**.
5. On the **Projects** resources page, choose the project that contains the running model that you want to stop.
6. In the **Models** section, choose the model that you want to stop.
7. On the model's details page, choose **Use model** and then choose **Integrate API to the cloud**.
8. Under **AWS CLI commands**, copy the AWS CLI command that calls `stop-model`.
9. At the command prompt, enter the `stop-model` command that you copied in the previous step. If you are using the `lookoutvision` profile to get credentials, add the `--profile lookoutvision-access` parameter.
10. At the console, choose **Models** in the left navigation page.
11. Check the **Status** column for the current status of the model. The model has stopped when the **Status** column value is **Training complete**.

Stopping your Amazon Lookout for Vision model (SDK)

You stop a model by calling the StopModel operation.

A model might take a while to stop. To check the current status, use `DescribeModel`.

To stop your model (SDK)

1. If you haven't already done so, install and configure the AWS CLI and the AWS SDKs. For more information, see Step 4: Set up the AWS CLI and AWS SDKs.
2. Use the following example code to stop a running model.

   CLI

   Change the following values:

   - `project-name` to the name of the project that contains the model that you want to stop.
   - `model-version` to the version of the model that you want to stop.

```
aws lookoutvision stop-model --project-name "project name"\
    --model-version model version \
  --profile lookoutvision-access
```

Python

This code is taken from the AWS Documentation SDK examples GitHub repository. See the full example here.

```
    @staticmethod
    def stop_model(lookoutvision_client, project_name, model_version):
        """
        Stops a running Lookout for Vision Model.

        :param lookoutvision_client: A Boto3 Lookout for Vision client.
        :param project_name: The name of the project that contains the version
of
                             the model that you want to stop hosting.
        :param model_version:  The version of the model that you want to stop
hosting.
        """
        try:
            logger.info("Stopping model version %s for %s", model_version,
project_name)
            response = lookoutvision_client.stop_model(
                ProjectName=project_name, ModelVersion=model_version
            )
            logger.info("Stopping hosting...")

            status = response["Status"]
            finished = False

            # Wait until stopped or failed.
            while finished is False:
                model_description = lookoutvision_client.describe_model(
                    ProjectName=project_name, ModelVersion=model_version
                )
                status = model_description["ModelDescription"]["Status"]

                if status == "STOPPING_HOSTING":
                    logger.info("Host stopping in progress...")
                    time.sleep(10)
                    continue

                if status == "TRAINED":
                    logger.info("Model is no longer hosted.")
                    finished = True
```

```
                    continue

                logger.info("Failed to stop model: %s ", status)
                finished = True

            if status != "TRAINED":
                logger.error("Error stopping model: %s", status)
                raise Exception(f"Error stopping model: {status}")
        except ClientError:
            logger.exception("Couldn't stop hosting model.")
            raise
```

Java V2

This code is taken from the AWS Documentation SDK examples GitHub repository. See the full example here.

```
/**
 * Stops the hosting an Amazon Lookout for Vision model. Returns when model has
 * stopped or if hosting fails.
 *
 * @param lfvClient   An Amazon Lookout for Vision client.
 * @param projectName The name of the project that contains the model that you
 *                    want to stop hosting.
 * @modelVersion The version of the model that you want to stop hosting.
 * @return ModelDescription The description of the model, which includes the
 *         model hosting status.
 */

public static ModelDescription stopModel(LookoutVisionClient lfvClient, String
 projectName,
                String modelVersion) throws LookoutVisionException,
 InterruptedException {

        logger.log(Level.INFO, "Stopping Model version {0} for project {1}.",
                        new Object[] { modelVersion, projectName });

        StopModelRequest stopModelRequest = StopModelRequest.builder()
                        .projectName(projectName)
                        .modelVersion(modelVersion)
                        .build();
```

```
        // Stop hosting the model.

        lfvClient.stopModel(stopModelRequest);

        DescribeModelRequest describeModelRequest =
DescribeModelRequest.builder()
                        .projectName(projectName)
                        .modelVersion(modelVersion)
                        .build();

        ModelDescription modelDescription = null;

        boolean finished = false;
        // Wait until model is stopped or failure occurs.
        do {

                modelDescription =
lfvClient.describeModel(describeModelRequest).modelDescription();

                switch (modelDescription.status()) {

                        case TRAINED:
                                logger.log(Level.INFO, "Model version {0} for
project {1} has stopped.",
                                                new Object[] { modelVersion,
projectName });
                                finished = true;
                                break;

                        case STOPPING_HOSTING:
                                logger.log(Level.INFO, "Model version {0} for
project {1} is stopping.",
                                                new Object[] { modelVersion,
projectName });

                                TimeUnit.SECONDS.sleep(60);

                                break;

                        default:
                                logger.log(Level.SEVERE,
                                                "Unexpected error when stopping
model version {0} for project {1}: {2}.",
```

```
                                                    new Object[] { projectName,
 modelVersion,

 modelDescription.status() });
                                finished = true;
                                break;

               }

       } while (!finished);

       logger.log(Level.INFO, "Finished stopping model version {0} for project
 {1} status: {2}",
                       new Object[] { modelVersion, projectName,
 modelDescription.statusMessage() });

       return modelDescription;

}
```

Detecting anomalies in an image

To detect anomalies in an image with a trained Amazon Lookout for Vision model, you call the DetectAnomalies operation. The result from `DetectAnomalies` includes a Boolean prediction that classifies the image as containing one or more anomalies and a confidence value for the prediction. If the model is an image segmentation model, the result also includes a colored mask showing the positions of different types of anomalies.

The images you supply to `DetectAnomalies` must have the same width and height dimensions as the images that you used to train the model.

`DetectAnomalies` accepts images as PNG or JPG format images. We recommend that the images are in the same encoding and compression format as those used to train the model. For example, if you train the model with PNG format images, call `DetectAnomalies` with PNG format images.

Before calling `DetectAnomalies`, you must first start your model with the `StartModel` operation. For more information, see Starting your Amazon Lookout for Vision model. You are charged for the amount of time, in minutes, that a model runs and for the number of anomaly detection units that your model uses. If you are not using a model, use the `StopModel` operation to stop your model. For more information, see Stopping your Amazon Lookout for Vision model.

Topics

- Calling DetectAnomalies
- Understanding the response from DetectAnomalies
- Determining if an image is anomalous
- Showing classification and segmentation information
- Finding anomalies with an AWS Lambda function

Calling DetectAnomalies

To call `DetectAnomalies`, specify the following:

- **Project** – The name of the project that contains the model that you want to use.
- **ModelVersion** – The version of the model that you want to use.
- **ContentType** – The type of image that you want analyze. Valid values are `image/png` (PNG format images) and `image/jpeg` (JPG format images).

- **Body** – The unencoded binary bytes that represent the image.

  The image must have the same dimensions as the images used to train the model.

The following example shows how to call `DetectAnomalies`. You can use the function response from the Python and Java examples to call functions in Determining if an image is anomalous.

AWS CLI

This AWS CLI command displays the JSON output for the `DetectAnomalies` CLI operation. Change the values of the following input parameters:

- `project name` with the name of the project that you want to use.
- `model version` with the version of the model that you want to use.
- `content type` with the type of the image that you want to use. Valid values are `image/png` (PNG format images) and `image/jpeg` (JPG format images).
- `file name` with the path and file name of the image that you want to use. Ensure that the file type matches the value of `content-type`.

```
aws lookoutvision detect-anomalies  --project-name project name\
    --model-version model version\
    --content-type content type\
    --body file name \
    --profile lookoutvision-access
```

Python

For the complete code example, see GitHub.

```
    def detect_anomalies(lookoutvision_client, project_name, model_version, photo):
        """
        Calls DetectAnomalies using the supplied project, model version, and image.
        :param lookoutvision_client: A Lookout for Vision Boto3 client.
        :param project: The project that contains the model that you want to use.
        :param model_version: The version of the model that you want to use.
        :param photo: The photo that you want to analyze.
        :return: The DetectAnomalyResult object that contains the analysis results.
        """
```

```
        image_type = imghdr.what(photo)
        if image_type == "jpeg":
            content_type = "image/jpeg"
        elif image_type == "png":
            content_type = "image/png"
        else:
            logger.info("Invalid image type for %s", photo)
            raise ValueError(
                f"Invalid file format. Supply a jpeg or png format file: {photo}")


        # Get images bytes for call to detect_anomalies
        with open(photo, "rb") as image:
            response = lookoutvision_client.detect_anomalies(
                ProjectName=project_name,
                ContentType=content_type,
                Body=image.read(),
                ModelVersion=model_version)

        return response['DetectAnomalyResult']
```

Java V2

```
  public static DetectAnomalyResult detectAnomalies(LookoutVisionClient lfvClient,
String projectName,
           String modelVersion,
           String photo) throws IOException, LookoutVisionException {
       /**
        * Creates an Amazon Lookout for Vision dataset from a manifest file.
        * Returns after Lookout for Vision creates the dataset.
        *
        * @param lfvClient    An Amazon Lookout for Vision client.
        * @param projectName  The name of the project in which you want to create a
        *                     dataset.
        * @param modelVersion The version of the model that you want to use.
        *
        * @param photo        The photo that you want to analyze.
        *
        * @return DetectAnomalyResult The analysis result from DetectAnomalies.
        */

       logger.log(Level.INFO, "Processing local file: {0}", photo);
```

```
        // Get image bytes.

        InputStream sourceStream = new FileInputStream(new File(photo));
        SdkBytes imageSDKBytes = SdkBytes.fromInputStream(sourceStream);
        byte[] imageBytes = imageSDKBytes.asByteArray();

        // Get the image type. Can be image/jpeg or image/png.
        String contentType = getImageType(imageBytes);

        // Detect anomalies in the supplied image.
        DetectAnomaliesRequest request =
DetectAnomaliesRequest.builder().projectName(projectName)
                .modelVersion(modelVersion).contentType(contentType).build();

        DetectAnomaliesResponse response = lfvClient.detectAnomalies(request,
                RequestBody.fromBytes(imageBytes));

        /*
         * Tip: You can also use the following to analyze a local file.
         * Path path = Paths.get(photo);
         * DetectAnomaliesResponse response = lfvClient.detectAnomalies(request,
path);
         */
        DetectAnomalyResult result = response.detectAnomalyResult();

        String prediction = "Prediction: Normal";

        if (Boolean.TRUE.equals(result.isAnomalous())) {
            prediction = "Prediction: Anomalous";
        }

        // Convert confidence to percentage.
        NumberFormat defaultFormat = NumberFormat.getPercentInstance();
        defaultFormat.setMinimumFractionDigits(1);
        String confidence = String.format("Confidence: %s",
defaultFormat.format(result.confidence()));

        // Log classification result.
        String photoPath = "File: " + photo;
        String[] imageLines = { photoPath, prediction, confidence };
        logger.log(Level.INFO, "Image: {0}\nAnomalous: {1}\nConfidence {2}",
imageLines);

        return result;
```

```
    }

    // Gets the image mime type. Supported formats are image/jpeg and image/png.
    private static String getImageType(byte[] image) throws IOException {

        InputStream is = new BufferedInputStream(new ByteArrayInputStream(image));
        String mimeType = URLConnection.guessContentTypeFromStream(is);

        logger.log(Level.INFO, "Image type: {0}", mimeType);

        if (mimeType.equals("image/jpeg") || mimeType.equals("image/png")) {
            return mimeType;
        }
        // Not a supported file type.
        logger.log(Level.SEVERE, "Unsupported image type: {0}", mimeType);
        throw new IOException(String.format("Wrong image type. %s format isn't
supported.", mimeType));
    }
```

Understanding the response from DetectAnomalies

The response from `DetectAnomalies` varies depending on the type of the model that you train (classification model or segmentation model). In both cases the response is a DetectAnomalyResult object.

Classification model

If your model is an Image classification model, the response from `DetectAnomalies` contains the following:

- **IsAnomalous**– A Boolean indicator that the image contains one or more anomalies.
- **Confidence**– The confidence that Amazon Lookout for Vision has in the accuracy of the anomaly prediction (`IsAnomalous`). `Confidence` is a floating point value between 0 and 1. A higher value indicates a higher confidence.
- **Source** – Information about the image passed to `DetectAnomalies`.

```
{
    "DetectAnomalyResult": {
```

```
        "Source": {
            "Type": "direct"
        },
        "IsAnomalous": true,
        "Confidence": 0.9996867775917053
    }
}
```

You determine if in an image is anomalous by checking the `IsAnomalous` field and confirming that the `Confidence` value is high enough for your needs.

If you're finding the confidence values returned by `DetectAnomalies` are too low, consider retraining the model. For example code, see Classification.

Segmentation model

If your model is an Image segmentation model, the response includes classification information and segmentation information, such as an image mask and anomaly types. Classification information is calculated separately from segmentation information and you shouldn't assume a relationship between them. If you don't get segmentation information in the response, check that you have the latest version of the AWS SDK installed (AWS Command Line Interface, if you are using the AWS CLI). For example code, see Segmentation and Showing classification and segmentation information.

- **IsAnomalous** (classification) – A Boolean indicator that classifies the image as either normal or anomalous.
- **Confidence** (classification) – The confidence that Amazon Lookout for Vision has in the accuracy of the classification of the image (`IsAnomalous`). `Confidence` is a floating point value between 0 and 1. A higher value indicates a higher confidence.
- **Source** – Information about the image passed to `DetectAnomalies`.
- **AnomalyMask** (segmentation) – A pixel mask covering anomalies found in the analyzed image. There can be multiple anomalies on the image. The color of a mask maps indicates the type of an anomaly. The mask colors map to the colors assigned to anomaly types in the training dataset. To find the anomaly type from a mask color, check `Color` in the `PixelAnomaly` field of each anomaly returned in the `Anomalies` list. For example code, see Showing classification and segmentation information.

- **Anomalies** (segmentation) – A list of anomalies found in the image. Each anomaly includes the anomaly type (`Name`), and pixel information (`PixelAnomaly`). `TotalPercentageArea` is the percentage area of the image that the anomaly covers. `Color` is the mask color for the anomaly.

  The first element in the list is always an anomaly type representing the image background (`BACKGROUND`) and shouldn't be considered an anomaly. Amazon Lookout for Vision automatically adds the background anomaly type to the response. You don't need to declare a background anomaly type in your dataset.

```
{
    "DetectAnomalyResult": {
        "Source": {
            "Type": "direct"
        },
        "IsAnomalous": true,
        "Confidence": 0.9996814727783203,
        "Anomalies": [
            {
                "Name": "background",
                "PixelAnomaly": {
                    "TotalPercentageArea": 0.998999834060669,
                    "Color": "#FFFFFF"
                }
            },
            {
                "Name": "scratch",
                "PixelAnomaly": {
                    "TotalPercentageArea": 0.0004034999874420464,
                    "Color": "#7ED321"
                }
            },
            {
                "Name": "dent",
                "PixelAnomaly": {
                    "TotalPercentageArea": 0.0005966666503809392,
                    "Color": "#4DD8FF"
                }
            }
        ],
        "AnomalyMask": "iVBORw0....."
    }
```

```
}
```

Determining if an image is anomalous

You can determine if an image is anomalous in a variety of ways. The method you choose depends on your use case and the type of your model. The following are potential solutions.

Topics

- Classification
- Segmentation

Classification

`IsAnomalous` classifies an image as anomalous, use the `Confidence` field to help decide if the image is actually anomalous. A higher value indicates greater confidence. For example, you might decide a product is defective only if the confidence is over 80%. You can classify images analyzed by classification models or by image segmentation models.

Python

For the complete code example, see GitHub.

```
    def reject_on_classification(image, prediction, confidence_limit):
        """
        Returns True if the anomaly confidence is greater than or equal to
        the supplied confidence limit.
        :param image: The name of the image file that was analyzed.
        :param prediction: The DetectAnomalyResult object returned from
DetectAnomalies
        :param confidence_limit: The minimum acceptable confidence. Float value
between 0 and 1.
        :return: True if the error condition indicates an anomaly, otherwise False.
        """

        reject = False

        logger.info("Checking classification for %s", image)

        if prediction['IsAnomalous'] and prediction['Confidence'] >=
confidence_limit:
```

```
            reject = True
            reject_info=(f"Rejected: Anomaly confidence
({prediction['Confidence']:.2%}) is greater"
                    f" than limit ({confidence_limit:.2%})")
            logger.info("%s", reject_info)

        if not reject:
            logger.info("No anomalies found.")
        return reject
```

Java V2

```
   public static boolean rejectOnClassification(String image, DetectAnomalyResult
prediction, float minConfidence) {
        /**
         * Rejects an image based on its anomaly classification and prediction
         * confidence
         *
         * @param image         The file name of the analyzed image.
         * @param prediction    The prediction for an image analyzed with
         *                      DetectAnomalies.
         * @param minConfidence The minimum acceptable confidence for the prediction
         *                      (0-1).
         *
         * @return boolean True if the image is anomalous, otherwise False.
         */

        Boolean reject = false;

        logger.log(Level.INFO, "Checking classification for {0}", image);

        String[] logParameters = { prediction.confidence().toString(),
String.valueOf(minConfidence) };

        if (Boolean.TRUE.equals(prediction.isAnomalous()) && prediction.confidence()
>= minConfidence) {
            logger.log(Level.INFO, "Rejected: Anomaly confidence {0} is greater than
confidence limit {1}",
                    logParameters);
            reject = true;
        }
        if (Boolean.FALSE.equals(reject))
            logger.log(Level.INFO, ": No anomalies found.");
```

```
        return reject;

    }
```

Segmentation

If your model is an image segmentation model, you can use the segmentation information to determine if an image contains anomalies. You can also use an image segmentation model to classify images. For example code that gets and display image masks, see Showing classification and segmentation information

Area of anomaly

Use the percentage coverage (`TotalPercentageArea`) of an anomaly on the image. For example, you might decide a product is defective if the area of an anomaly is greater than 1% of the image.

Python

For the complete code example, see GitHub.

```
   def reject_on_coverage(image, prediction, confidence_limit, anomaly_label,
coverage_limit):
       """
       Checks if the coverage area of an anomaly is greater than the coverage limit
and if
       the prediction confidence is greater than the confidence limit.
       :param image: The name of the image file that was analyzed.
       :param prediction: The DetectAnomalyResult object returned from
DetectAnomalies
       :param confidence_limit: The minimum acceptable confidence (float 0-1).
       :anomaly_label: The anomaly label for the type of anomaly that you want to
check.
       :coverage_limit: The maximum acceptable percentage coverage of an anomaly
(float 0-1).
       :return: True if the error condition indicates an anomaly, otherwise False.
       """

       reject = False

       logger.info("Checking coverage for %s", image)
```

```
        if prediction['IsAnomalous'] and prediction['Confidence'] >=
 confidence_limit:
            for anomaly in prediction['Anomalies']:
                if (anomaly['Name'] == anomaly_label and
                        anomaly['PixelAnomaly']['TotalPercentageArea'] >
 (coverage_limit)):
                    reject = True
                    reject_info=(f"Rejected: Anomaly confidence
 ({prediction['Confidence']:.2%}) "
                        f"is greater than limit ({confidence_limit:.2%}) and
 {anomaly['Name']} "
                        f"coverage ({anomaly['PixelAnomaly']
['TotalPercentageArea']:.2%}) "
                        f"is greater than limit ({coverage_limit:.2%})")

                    logger.info("%s", reject_info)

        if not reject:
            logger.info("No anomalies found.")

        return reject
```

Java V2

```
   public static Boolean rejectOnCoverage(String image, DetectAnomalyResult
 prediction, float minConfidence,
            String anomalyType, float maxCoverage) {
        /**
         * Rejects an image based on a maximum allowable coverage area for an
 anomaly
         * type.
         *
         * @param image         The file name of the analyzed image.
         * @param prediction    The prediction for an image analyzed with
         *                      DetectAnomalies.
         * @param minConfidence The minimum acceptable confidence for the prediction
         *                      (0-1).
         * @param anomalyTypes  The anomaly type to check.
         * @param maxCoverage   The maximum allowable coverage area of the anomaly
 type.
         *                      (0-1).
```

```
         *
         * @return boolean True if the coverage area of the anomaly type exceeds the
         *         maximum allowed, otherwise False.
         */

        Boolean reject = false;

        logger.log(Level.INFO, "Checking coverage for {0}", image);

        if (Boolean.TRUE.equals(prediction.isAnomalous()) && prediction.confidence()
>= minConfidence) {
            for (Anomaly anomaly : prediction.anomalies()) {

                if (Objects.equals(anomaly.name(), anomalyType)
                        && anomaly.pixelAnomaly().totalPercentageArea() >=
maxCoverage) {

                    String[] logParameters = { prediction.confidence().toString(),
                            String.valueOf(minConfidence),

String.valueOf(anomaly.pixelAnomaly().totalPercentageArea()),
                            String.valueOf(maxCoverage) };
                    logger.log(Level.INFO,
                            "Rejected: Anomaly confidence {0} is greater than
confidence limit {1} and " +
                                    "{2} anomaly type coverage is higher than
coverage limit {3}\n",
                            logParameters);
                    reject = true;

                }
            }
        }

        if (Boolean.FALSE.equals(reject))
            logger.log(Level.INFO, ": No anomalies found.");

        return reject;
    }
```

Number of anomaly types

Use a count of different anomaly types (`Name`) found on the image. For example, you might decide a product is defective if there is more than two types of anomaly present.

Python

For the complete code example, see GitHub.

```
   def reject_on_anomaly_types(image, prediction, confidence_limit,
anomaly_types_limit):
       """
       Checks if the number of anomaly types is greater than than the anomaly types
       limit and if the prediction confidence is greater than the confidence limit.
       :param image: The name of the image file that was analyzed.
       :param prediction: The DetectAnomalyResult object returned from
DetectAnomalies
       :param confidence: The minimum acceptable confidence. Float value between 0
and 1.
       :param anomaly_types_limit: The maximum number of allowable anomaly types
(Integer).
       :return: True if the error condition indicates an anomaly, otherwise False.
       """

       logger.info("Checking number of anomaly types for %s",image)

       reject = False

       if prediction['IsAnomalous'] and prediction['Confidence'] >=
confidence_limit:

           anomaly_types = {anomaly['Name'] for anomaly in prediction['Anomalies']\
               if anomaly['Name'] != 'background'}

           if len (anomaly_types) > anomaly_types_limit:
                   reject = True
                   reject_info = (f"Rejected: Anomaly confidence
({prediction['Confidence']:.2%}) "
                   f"is greater than limit ({confidence_limit:.2%}) and "
                   f"the number of anomaly types ({len(anomaly_types)-1}) is "
                   f"greater than the limit ({anomaly_types_limit})")

                   logger.info("%s", reject_info)
```

```
        if not reject:
            logger.info("No anomalies found.")
        return reject
```

Java V2

```
   public static Boolean rejectOnAnomalyTypeCount(String image, DetectAnomalyResult
prediction,
           float minConfidence, Integer maxAnomalyTypes) {

       /**
        * Rejects an image based on a maximum allowable number of anomaly types.
        *
        * @param image           The file name of the analyzed image.
        * @param prediction      The prediction for an image analyzed with
        *                        DetectAnomalies.
        * @param minConfidence   The minimum acceptable confidence for the
predictio
        *                        (0-1).
        * @param maxAnomalyTypes The maximum allowable number of anomaly types.
        *
        * @return boolean True if the image contains more than the maximum allowed
        *         anomaly types, otherwise False.
        */

       Boolean reject = false;

       logger.log(Level.INFO, "Checking coverage for {0}", image);

       Set<String> defectTypes = new HashSet<>();

       if (Boolean.TRUE.equals(prediction.isAnomalous()) && prediction.confidence()
>= minConfidence) {
           for (Anomaly anomaly : prediction.anomalies()) {
               defectTypes.add(anomaly.name());
           }
           // Reduce defect types by one to account for 'background' anomaly type.
           if ((defectTypes.size() - 1) > maxAnomalyTypes) {
               String[] logParameters = { prediction.confidence().toString(),
                       String.valueOf(minConfidence),
                       String.valueOf(defectTypes.size()),
                       String.valueOf(maxAnomalyTypes) };
```

```
                logger.log(Level.INFO, "Rejected: Anomaly confidence {0} is >=
minimum confidence {1} and " +
                        "the number of anomaly types {2} > the allowable number of
anomaly types {3}\n", logParameters);
                reject = true;
            }

        }

        if (Boolean.FALSE.equals(reject))
            logger.log(Level.INFO, ": No anomalies found.");

        return reject;
    }
```

Showing classification and segmentation information

This example shows the analyzed image and overlays the analysis results. If the response includes an anomaly mask, the mask is shown in the colors of the associated anomaly types.

To show image classification and image segmentation information

1. If you haven't already done so, do the following:

   a. If you haven't already done so, install and configure the AWS CLI and the AWS SDKs. For more information, see Step 4: Set up the AWS CLI and AWS SDKs.

   b. Train your model.

   c. Start your model.

2. Make sure the user calling `DetectAnomalies` has access to the model version that you want to use. For more information, see Set up SDK permissions.

3. Use the following code.

   Python

   The following example code detects anomalies in an image that you supply. The example takes the following command line options:

   - `project` – the name of the project that you want to use.
   - `version` – the version of the model, within the project, that you want to use.

- image – the path and file of a local image file (JPEG or PNG format).

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
Shows how to detect and show anomalies in an image using a trained Amazon
 Lookout
for Vision model. The script displays the analysed image and overlays mask and
 analysis
output.
"""

import argparse
import logging
import io
import boto3
from PIL import Image, ImageDraw, ImageFont

from botocore.exceptions import ClientError

logger = logging.getLogger(__name__)


class ShowAnomalies:
    """
    Class to detect and show anomalies in an image analyzed by detect_anomalies.
    """



    @staticmethod
    def draw_line(draw, text, fnt,  y_coordinate, color):
        """
        Draws a line of text on the supplied drawing surface.
        :param draw: The surface on which to draw the text.
        :param text: The text to draw in the drawing surface.
        :param fnt: The font for the text.
        :param y_coordinate: The y position for the text.
        :param color: The color for the text.
        :returns The y coordinate for the next line of text.
        """
        text_width, text_height = draw.textsize(text, fnt)
        draw.rectangle([(10, y_coordinate), (text_width + 10,
```

```
                                                y_coordinate + text_height)],
 fill="black")
        draw.text((10, y_coordinate), text, fill=color, font=fnt)

        y_coordinate += text_height

        return y_coordinate

    @staticmethod
    def draw_analysis_text(image, analysis):
        """
        Draws classification and segmentation info onto supplied image
        overlay analysis results on an image analyzed by detect_anomalies.
        :param analysis: The response from a call to detect_anomalies.
        :returns Nothing
        """

        ## Calculate a reasonable font size based on image width.
        font_size = int(image.size[0]/32)

        fnt = ImageFont.truetype('/Library/Fonts/Tahoma.ttf', font_size)

        draw = ImageDraw.Draw(image)

        y_coordinate = 0

        # Draw classification information.
        prediction = "Anomalous" if analysis["DetectAnomalyResult"]
["IsAnomalous"] \
            else "Normal"

        confidence = analysis["DetectAnomalyResult"]["Confidence"]
        found_anomalies = analysis["DetectAnomalyResult"]['Anomalies']
        segmentation_info = False

        logger.info("Prediction: %s", format(prediction))
        logger.info("Confidence: %s", format(confidence))

        y_coordinate = 0
        y_coordinate = ShowAnomalies.draw_line(
            draw, "Classification", fnt, y_coordinate, "white")
        y_coordinate = ShowAnomalies.draw_line(
            draw, f"  Prediction: {prediction}", fnt, y_coordinate, "white")
        y_coordinate = ShowAnomalies.draw_line(
```

```
            draw, f"  Confidence: {confidence:.2%}", fnt, y_coordinate, "white")

        # Draw segmentation information, if present.
        if (len(found_anomalies)) > 1:
            logger.info("Anomalies:")

            y_coordinate = ShowAnomalies.draw_line(
                draw, "Segmentation:", fnt, y_coordinate, "white")
            for i in range(1, len(found_anomalies)):

                # Only display info if more than 0% coverage found.
                percent_coverage = found_anomalies[i]['PixelAnomaly']
['TotalPercentageArea']
                if percent_coverage > 0:
                    segmentation_info = True
                    logger.info("  %s", found_anomalies[i]['Name'])
                    logger.info("    Color: %s",
                                found_anomalies[i]['PixelAnomaly']['Color'])
                    logger.info("    Area: %s", percent_coverage)
                    y_coordinate = ShowAnomalies.draw_line(
                        draw,
                        f"  Anomaly: {found_anomalies[i]['Name']}. Area:
 {percent_coverage:.2%}",
                        fnt,
                        y_coordinate,
                        found_anomalies[i]['PixelAnomaly']['Color'])

            if not segmentation_info:
                y_coordinate = ShowAnomalies.draw_line(
                    draw, "No segmentation information found.", fnt,
 y_coordinate, "white")




    @staticmethod
    def show_anomaly_prediction(lookoutvision_client, project_name,
 model_version, photo):
        """
        Detects anomalies in an image (jpg/png) by using your Amazon Lookout for
 Vision
        model. Displays the image and overlays prediction information text.
        :param lookoutvision_client: An Amazon Lookout for Vision Boto3 client.
```

```
        :param project_name: The name of the project that contains the model
that
        you want to use.
        :param model_version: The version of the model that you want to use.
        :param photo: The path and name of the image in which you want to detect
        anomalies.
        """
        try:

            logger.info("Detecting anomalies in %s", photo)

            image = Image.open(photo)
            image_type = Image.MIME[image.format]

            # Check that image type is valid.
            if image_type not in ("image/jpeg", "image/png"):
                logger.info("Invalid image type for %s", photo)
                raise ValueError(
                    f"Invalid file format. Supply a jpeg or png format file:
{photo}"
                )

            # Get images bytes for call to detect_anomalies.
            image_bytes = io.BytesIO()
            image.save(image_bytes, format=image.format)
            image_bytes = image_bytes.getvalue()

            # Analyze the image.
            response = lookoutvision_client.detect_anomalies(
                ProjectName=project_name,
                ContentType=image_type,
                Body=image_bytes,
                ModelVersion=model_version
            )

            # Overlay mask onto analyzed image.
            image_mask_bytes = response["DetectAnomalyResult"]["AnomalyMask"]
            image_mask = Image.open(io.BytesIO(image_mask_bytes))

            final_img = Image.blend(image, image_mask, 0.5) \
                if response["DetectAnomalyResult"]["IsAnomalous"] else image

            # Overlay analysis output on image.
            ShowAnomalies.draw_analysis_text(final_img, response)
```

```
            final_img.show()

        except ClientError as err:
            logger.info(format(err))
            raise


def add_arguments(parser):
    """
    Adds command line arguments to the parser.
    :param parser: The command line parser.
    """

    parser.add_argument(
        "project", help="The project containing the model that you want to use."
    )
    parser.add_argument(
        "version", help="The version of the model that you want to use."
    )
    parser.add_argument(
        "image",
        help="The file that you want to analyze. "
        "Supply a local file path.",
    )


def main():
    """
    Entrypoint for anomaly detection example.
    """

    try:
        logging.basicConfig(level=logging.INFO,
                            format="%(levelname)s: %(message)s")

        session = boto3.Session(
            profile_name='lookoutvision-access')

        lookoutvision_client = session.client("lookoutvision")

        parser = argparse.ArgumentParser(usage=argparse.SUPPRESS)

        add_arguments(parser)
```

```
        args = parser.parse_args()

        # Analyze the image and show results.
        ShowAnomalies.show_anomaly_prediction(
            lookoutvision_client, args.project, args.version, args.image
        )

    except ClientError as err:
        print("A service error occured: " +
              format(err.response["Error"]["Message"]))
    except FileNotFoundError as err:
        print("The supplied file couldn't be found: " + err.filename)
    except ValueError as err:
        print("A value error occured. " + format(err))
    else:
        print("Successfully completed analysis.")


if __name__ == "__main__":
    main()
```

Java 2

The following example code detects anomalies in an image that you supply. The example takes the following command line options:

- `project` – the name of the project that you want to use.
- `version` – the version of the model, within the project, that you want to use.
- `image` – the path and file of a local image file (JPEG or PNG format).

```
/*
   Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
   SPDX-License-Identifier: Apache-2.0
*/


package com.example.lookoutvision;

import software.amazon.awssdk.auth.credentials.ProfileCredentialsProvider;
import software.amazon.awssdk.core.SdkBytes;
```

```
import software.amazon.awssdk.core.sync.RequestBody;
import software.amazon.awssdk.services.lookoutvision.LookoutVisionClient;
import software.amazon.awssdk.services.lookoutvision.model.Anomaly;
import
 software.amazon.awssdk.services.lookoutvision.model.DetectAnomaliesRequest;
import
 software.amazon.awssdk.services.lookoutvision.model.DetectAnomaliesResponse;
import software.amazon.awssdk.services.lookoutvision.model.DetectAnomalyResult;
import
 software.amazon.awssdk.services.lookoutvision.model.LookoutVisionException;

import java.io.BufferedInputStream;
import java.io.ByteArrayInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.net.URLConnection;

import java.text.NumberFormat;
import java.awt.*;
import java.awt.font.LineMetrics;
import java.awt.image.BufferedImage;
import javax.imageio.ImageIO;
import javax.swing.*;

import java.util.logging.Level;
import java.util.logging.Logger;

// Finds anomalies on a supplied image.
public class ShowAnomalies extends JPanel {
/**
 * Finds and displays anomalies on a supplied image.
 */

    private static final long serialVersionUID = 1L;
    private transient BufferedImage image;
    private transient BufferedImage maskImage;
    private transient Dimension dimension;
    public static final Logger logger =
 Logger.getLogger(ShowAnomalies.class.getName());

    // Constructor. Finds anomalies in a local image file.
```

```
   public ShowAnomalies(LookoutVisionClient lfvClient, String projectName,
String modelVersion,
           String photo) throws IOException, LookoutVisionException {

       logger.log(Level.INFO, "Processing local file: {0}", photo);

       maskImage = null;

       // Get image bytes and buffered image.
       InputStream sourceStream = new FileInputStream(new File(photo));
       SdkBytes imageSDKBytes = SdkBytes.fromInputStream(sourceStream);
       byte[] imageBytes = imageSDKBytes.asByteArray();
       ByteArrayInputStream inputStream = new
ByteArrayInputStream(imageSDKBytes.asByteArray());
       image = ImageIO.read(inputStream);

       // Get the image type. Can be image/jpeg or image/png.
       String contentType = getImageType(imageBytes);

       // Set the size of the window that shows the image.
       setWindowDimensions();

       // Detect anomalies in the supplied image.
       DetectAnomaliesRequest request =
DetectAnomaliesRequest.builder().projectName(projectName)
               .modelVersion(modelVersion).contentType(contentType).build();

       DetectAnomaliesResponse response = lfvClient.detectAnomalies(request,
               RequestBody.fromBytes(imageBytes));

       /*
        * Tip: You can also use the following to analyze a local file.
        * Path path = Paths.get(photo);
        * DetectAnomaliesResponse response = lfvClient.detectAnomalies(request,
path);
        */
       DetectAnomalyResult result = response.detectAnomalyResult();


       if (result.anomalyMask() != null){
           SdkBytes maskSDKBytes = result.anomalyMask();

           ByteArrayInputStream maskInputStream = new
ByteArrayInputStream(maskSDKBytes.asByteArray());
```

```
            maskImage = ImageIO.read(maskInputStream);
        }

        drawImageInfo(result);

    }

    // Sets window dimensions to 1/2 screen size, unless image is smaller.
    public void setWindowDimensions() {
        dimension = java.awt.Toolkit.getDefaultToolkit().getScreenSize();

        dimension.width = (int) dimension.getWidth() / 2;
        dimension.height = (int) dimension.getHeight() / 2;

        if (image.getWidth() < dimension.width || image.getHeight() <
 dimension.height) {
            dimension.width = image.getWidth();
            dimension.height = image.getHeight();
        }
        setPreferredSize(dimension);

    }

    private int drawLine(Graphics2D g2d, String line, FontMetrics metrics, int
 yPos, Color color) {
    /**
    * Draws a line of text at the spsecified y position and color.
    * confidence
    *
    * @param g2D The Graphics2D object for the image.
    * @param line The line of text to draw.
    * @param metrics The font information to use.
    * @param yPos The y position for the line of text.
    *
    * @return  The yPos for the next line of text.
    */

        int indent = 10;

        // Get text height, width, and descent.
        int textWidth = metrics.stringWidth(line);
        LineMetrics lm = metrics.getLineMetrics(line, g2d);
        int textHeight = (int) lm.getHeight();
        int descent = (int) lm.getDescent();
```

```
        int y2Pos = (yPos + textHeight) - descent;

        // Draw black rectangle.
        g2d.setColor(Color.BLACK);
        g2d.fillRect(indent, yPos, textWidth, textHeight);

        // Draw text.
        g2d.setColor(color);
        g2d.drawString(line, indent, y2Pos);

        yPos += textHeight;

        return yPos;

    }

    public void drawImageInfo(DetectAnomalyResult result) {
    /**
     * Draws the results from DetectAnomalies onto the output image.
     *
     * @param result The response from a call to
     *               DetectAnomalies.
     *
     */

        // Set up drawing.
        Graphics2D g2d = image.createGraphics();

        if (result.anomalyMask() != null){
            Composite composite = g2d.getComposite();
            g2d.setComposite(AlphaComposite.SrcOver.derive(0.5f));
            int x = (image.getWidth() - maskImage.getWidth()) / 2;
            int y = (image.getHeight() - maskImage.getHeight()) / 2;
            g2d.drawImage(maskImage, x, y, null);
            // Set composite for overlaying text.
            g2d.setComposite(composite);
        }

        //Calculate font size based on arbitary 32 pixel image width.
        int fontSize = (image.getWidth() / 32);


        g2d.setFont(new Font("Tahoma", Font.PLAIN, fontSize));
```

```
        Font font = g2d.getFont();
        FontMetrics metrics = g2d.getFontMetrics(font);

        // Get classification information.

        String prediction = "Prediction: Normal";

        if (Boolean.TRUE.equals(result.isAnomalous())) {
            prediction = "Prediction: Anomalous";
        }

        // Convert prediction to percentage.
        NumberFormat defaultFormat = NumberFormat.getPercentInstance();
        defaultFormat.setMinimumFractionDigits(1);
        String confidence = String.format("Confidence: %s",
defaultFormat.format(result.confidence()));

        // Draw classification information.
        int yPos = 0;

        yPos = drawLine(g2d, "Classification:", metrics, yPos, Color.WHITE);
        yPos = drawLine(g2d, prediction, metrics, yPos, Color.WHITE);
        yPos = drawLine(g2d, confidence, metrics, yPos, Color.WHITE);

        // Draw segmentation info.
        yPos = drawLine(g2d, "Segmentation:", metrics, yPos, Color.WHITE);

        // Ignore background label, so size must be > 1
        if (result.anomalies().size() > 1) {
            for (Anomaly anomaly : result.anomalies()) {
                if (anomaly.name().equals("background"))
                    continue;
                String label = String.format("Anomaly: %s. Area: %s",
anomaly.name(),

defaultFormat.format(anomaly.pixelAnomaly().totalPercentageArea()));
                Color anomalyColor =
Color.decode((anomaly.pixelAnomaly().color()));
                yPos = drawLine(g2d, label, metrics, yPos, anomalyColor);

            }

        } else {
            drawLine(g2d, "None found.", metrics, yPos, Color.WHITE);
```

```
        }

        g2d.dispose();

    }

    @Override
    public void paintComponent(Graphics g)
    /**
     * Draws the image and analysis results.
     *
     * @param g The Graphics context object for drawing.
     *          DetectAnomalies.
     *
     */
    {

        Graphics2D g2d = (Graphics2D) g; // Create a Java2D version of g.

        // Draw the image.
        g2d.drawImage(image, 0, 0, dimension.width, dimension.height, this);

    }

    // Gets the image mime type. Supported formats are image/jpeg and image/png.

    private String getImageType(byte[] image) throws IOException
    /**
     * Gets the file type of a supplied image. Raises an exception if the image
     * isn't compatible with with Amazon Lookout for Vision.
     *
     * @param image The image that you want to check.
     *
     * @return String The type of the image.
     */

    {
        InputStream is = new BufferedInputStream(new
ByteArrayInputStream(image));
        String mimeType = URLConnection.guessContentTypeFromStream(is);

        logger.log(Level.INFO, "Image type: {0}", mimeType);
```

```
        if (mimeType.equals("image/jpeg") || mimeType.equals("image/png")) {
            return mimeType;
        }
        // Not a supported file type.
        logger.log(Level.SEVERE, "Unsupported image type: {0}", mimeType);
        throw new IOException(String.format("Wrong image type. %s format isn't
supported.", mimeType));
    }

    public static void main(String[] args) throws Exception {

        String photo = null;
        String projectName = null;
        String modelVersion = null;

        final String USAGE = "\n" +
                "Usage:\n" +
                "    DetectAnomalies <project> <version> <image> \n\n" +
                "Where:\n" +
                "    project - The Lookout for Vision project.\n\n" +
                "    version - The version of the model within the project.\n\n"
+
                "    image - The path and filename of a local image. \n\n";

        try {

            if (args.length != 3) {
                System.out.println(USAGE);
                System.exit(1);
            }

            projectName = args[0];
            modelVersion = args[1];
            photo = args[2];
            ShowAnomalies panel = null;

            // Get the Lookout for Vision client.
            LookoutVisionClient lfvClient = LookoutVisionClient.builder()

.credentialsProvider(ProfileCredentialsProvider.create("lookoutvision-access"))
                .build();

            // Create frame and panel.
            JFrame frame = new JFrame(photo);
```

```
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

            panel = new ShowAnomalies(lfvClient, projectName, modelVersion,
 photo);

            frame.setContentPane(panel);
            frame.pack();
            frame.setVisible(true);


        } catch (LookoutVisionException lfvError) {
            logger.log(Level.SEVERE, "Lookout for Vision client error: {0}:
 {1}",
                    new Object[] { lfvError.awsErrorDetails().errorCode(),
                            lfvError.awsErrorDetails().errorMessage() });
            System.out.println(String.format("lookout for vision client error:
 %s", lfvError.getMessage()));
            System.exit(1);

        } catch (FileNotFoundException fileError) {
            logger.log(Level.SEVERE, "Could not find file: {0}",
 fileError.getMessage());
            System.out.println(String.format("Could not find file: %s",
 fileError.getMessage()));
            System.exit(1);

        } catch (IOException ioError) {
            logger.log(Level.SEVERE, "IO error {0}", ioError.getMessage());
            System.out.println(String.format("IO error: %s",
 ioError.getMessage()));
            System.exit(1);
        }

    }
}
```

4. If you aren't planning to continue using your model, stop your model.

Finding anomalies with an AWS Lambda function

AWS Lambda is a compute service that lets you run code without provisioning or managing servers. For example, you can analyze images submitted from a mobile application without having to create a server to host the application code. The following instructions show how to create a Lambda function in Python that calls DetectAnomalies. The function analyzes a supplied image and returns a classification for the presence of anomalies in that image. The instructions include example Python code showing how to call the Lambda function with an image in an Amazon S3 bucket, or an image supplied from a local computer.

Topics

- Step 1: Create an AWS Lambda function (console)
- Step 2: (Optional) Create a layer (console)
- Step 3: Add Python code (console)
- Step 4: Try your Lambda function

Step 1: Create an AWS Lambda function (console)

In this step, you create an empty AWS function and an IAM execution role that lets your function call the `DetectAnomalies` operation. It also grants access to the Amazon S3 bucket that stores images for analysis. You also specify environment variables for the following:

- The Amazon Lookout for Vision project and model version that you want your Lambda function to use.
- The confidence limit that you want the model to use.

Later you add the source code and optionally a layer to the Lambda function.

To create an AWS Lambda function (console)

1. Sign in to the AWS Management Console and open the AWS Lambda console at https://console.aws.amazon.com/lambda/.
2. Choose **Create function**. For more information, see Create a Lambda Function with the Console.
3. Choose the following options.

- Choose **Author from scratch**.
- Enter a value for **Function name**.
- For **Runtime** choose **Python 3.10**.

4. Choose **Create function** to create the AWS Lambda function.
5. On the function page, Choose the **Configuration** tab.
6. On the **Environment variables** pane, choose **Edit**.
7. Add the following environment variables. For each variable choose **Add enviroment variable** and then enter the variable key and value.

Key	Value
PROJECT_NAME	The Lookout for Vision project that contains the model you want to use.
MODEL_VERSION	The version of the model that you want to use.
CONFIDENCE	The minumum value (0-100) for the model's confidence that the prediction is anomalous . If the confidence is lower, the classification is deemed normal.

8. Choose **Save** to save the environment variables.
9. On the **Permissions** pane, Under **Role name**, choose the execution role to open the role in the IAM console.
10. In the **Permissions** tab, choose **Add permissions** and then **Create inline policy**.
11. Choose **JSON** and replace the existing policy with the following policy.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": "lookoutvision:DetectAnomalies",
            "Resource": "*",
            "Effect": "Allow",
            "Sid": "DetectAnomaliesAccess"
```

```
        }
    ]
}
```

12. Choose **Next**.
13. In **Policy details**, enter a name for the policy, such as *DetectAnomalies-access*.
14. Choose **Create policy**.
15. If you are storing images for analysis in an Amazon S3 bucket, repeat steps 10–14.

    a. For step 11, use the following policy. Replace *bucket/folder path* with the Amazon S3 bucket and folder path to the images that you want to analyze.

    ```
    {
        "Version": "2012-10-17",
        "Statement": [
            {
                "Sid": "S3Access",
                "Effect": "Allow",
                "Action": "s3:GetObject",
                "Resource": "arn:aws:s3:::bucket/folder path/*"
            }
        ]
    }
    ```

    b. For step 13, choose a different policy name, such as *S3Bucket-access*.

Step 2: (Optional) Create a layer (console)

To run this example, You don't need to do this step. The `DetectAnomalies` operation is included in the default Lambda Python environment as part of AWS SDK for Python (Boto3). If other parts of your Lambda function need recent AWS service updates that aren't in the default Lambda Python environment, do this step to add the latest Boto3 SDK release as a layer to your function.

First, you create a .zip file archive that contains the Boto3 SDK. You then create a layer and add the .zip file archive to the layer. For more information, see Using layers with your Lambda function.

To create and add a layer (console)

1. Open a command prompt and enter the following commands.

```
pip install boto3 --target python/.
zip boto3-layer.zip -r python/
```

2. Note the name of the zip file (boto3-layer.zip). You need it in step 6 of this procedure.
3. Open the AWS Lambda console at https://console.aws.amazon.com/lambda/.
4. In the navigation pane, choose **Layers**.
5. Choose **Create layer**.
6. Enter values for **Name** and **Description**.
7. Choose **Upload a .zip file** and choose **Upload**.
8. In the dialog box, choose the .zip file archive (boto3-layer.zip) that you created in step 1 of this procedure.
9. For compatible runtimes, choose **Python 3.9**.
10. Choose **Create** to create the layer.
11. Choose the navigation pane menu icon.
12. In the navigation pane, choose **Functions**.
13. In the resources list, choose the function that you created in Step 1: Create an AWS Lambda function (console).
14. Choose the **Code** tab.
15. In the **Layers** section, choose **Add a layer**.
16. Choose **Custom layers**.
17. In **Custom layers**, choose the layer name that you entered in step 6.
18. In **Version** choose the layer version, which should be 1.
19. Choose **Add**.

Step 3: Add Python code (console)

In this step, you add Python code to your Lambda function by using the Lambda console code editor. The code analyzes a supplied image with `DetectAnomalies` and returns a classification (true if the image is anomalous, false if the image is normal). The supplied image can be located in an Amazon S3 bucket or provided as byte64 encoded image bytes.

To add Python code (console)

1. If you're not in the Lambda console, do the following:
   a. Open the AWS Lambda console at https://console.aws.amazon.com/lambda/.
   b. Open the Lambda function you created in Step 1: Create an AWS Lambda function (console).
2. Choose the **Code** tab.
3. In **Code source**, replace the code in **lambda_function.py** with the following:

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

"""
Purpose
An AWS lambda function that analyzes images with an Amazon Lookout for Vision
 model.
"""
import base64
import imghdr
from os import environ
from io import BytesIO
import logging
import boto3

from botocore.exceptions import ClientError


logger = logging.getLogger(__name__)


# Get the model and confidence.
project_name = environ['PROJECT_NAME']
model_version = environ['MODEL_VERSION']
min_confidence = int(environ.get('CONFIDENCE', 50))

lookoutvision_client = boto3.client('lookoutvision')


def lambda_handler(event, context):
    """
    Lambda handler function
```

```
    param: event: The event object for the Lambda function.
    param: context: The context object for the lambda function.
    return: The labels found in the image passed in the event
    object.
    """

    try:

        file_name = ""

        # Determine image source.
        if 'image' in event:
            # Decode the encoded image
            image_bytes = event['image'].encode('utf-8')
            img_b64decoded = base64.b64decode(image_bytes)
            image_type = get_image_type(img_b64decoded)
            image = BytesIO(img_b64decoded)
            file_name = event['filename']

        elif 'S3Object' in event:
            bucket = boto3.resource('s3').Bucket(event['S3Object']['Bucket'])
            image_object = bucket.Object(event['S3Object']['Name'])
            image = image_object.get().get('Body').read()
            image_type = get_image_type(image)
            file_name = f"s3://{event['S3Object']['Bucket']}/{event['S3Object']
['Name']}"

        else:
            raise ValueError(
                'Invalid image source. Only base 64 encoded image bytes or images
 in S3 buckets are supported.')

        # Analyze the image.
        response = lookoutvision_client.detect_anomalies(
            ProjectName=project_name,
            ContentType=image_type,
            Body=image,
            ModelVersion=model_version)

        reject = reject_on_classification(
            response['DetectAnomalyResult'],
 confidence_limit=float(environ['CONFIDENCE'])/100)

        status = "anomalous" if reject else "normal"
```

```
        lambda_response = {
            "statusCode": 200,
            "body": {
                "Reject": reject,
                "RejectMessage": f"Image {file_name} is {status}."
            }
        }

    except ClientError as err:
        error_message = f"Couldn't analyze {file_name}. " + \
            err.response['Error']['Message']

        lambda_response = {
            'statusCode': 400,
            'body': {
                "Error": err.response['Error']['Code'],
                "ErrorMessage": error_message,
                "Image": file_name
            }
        }
        logger.error("Error function %s: %s",
                     context.invoked_function_arn, error_message)

    except ValueError as val_error:

        lambda_response = {
            'statusCode': 400,
            'body': {
                "Error": "ValueError",
                "ErrorMessage": format(val_error),
                "Image": event['filename']
            }
        }
        logger.error("Error function %s: %s",
                     context.invoked_function_arn, format(val_error))

    return lambda_response

def get_image_type(image):
    """
    Gets the format of the image. Raises an error
    if the type is not PNG or JPEG.
    :param image: The image that you want to check.
```

```
    :return The type of the image.

    """
    image_type = imghdr.what(None, image)

    if image_type == "jpeg":
        content_type = "image/jpeg"
    elif image_type == "png":
        content_type = "image/png"
    else:
        logger.info("Invalid image type")
        raise ValueError(
            "Invalid file format. Supply a jpeg or png format file.")
    return content_type


def reject_on_classification(prediction, confidence_limit):
    """
    Returns True if the anomaly confidence is greater than or equal to
    the supplied confidence limit.
    :param image: The name of the image file that was analyzed.
    :param prediction: The DetectAnomalyResult object returned from DetectAnomalies
    :param confidence_limit: The minimum acceptable confidence. Float value between
 0 and 1.
    :return: True if the error condition indicates an anomaly, otherwise False.
    """

    reject = False

    if prediction['IsAnomalous'] and prediction['Confidence'] >= confidence_limit:
        reject = True
        reject_info = (f"Rejected: Anomaly confidence
 ({prediction['Confidence']:.2%}) is greater"
                       f" than limit ({confidence_limit:.2%})")
        logger.info("%s", reject_info)

    if not reject:
        logger.info("No anomalies found.")
    return reject
```

4. Choose **Deploy** to deploy your Lambda function.

Step 4: Try your Lambda function

In this step you use Python code on your computer to pass a local image, or an image in an Amazon S3 bucket, to your Lambda function. Images passed from a local computer must be smaller than 6291456 bytes. If your images are larger, upload the images to an Amazon S3 bucket and call the script with the Amazon S3 path to the image. For information about uploading image files to an Amazon S3 bucket, see Uploading objects.

Make sure you run the code in the same AWS Region in which you created the Lambda function. You can view the AWS Region for your Lambda function in the navigation bar of the function details page in the Lambda console.

If the AWS Lambda function returns a timeout error, extend the timeout period for the Lambda function function, For more information, see Configuring function timeout (console).

For more information about invoking a Lambda function from your code, see Invoking AWS Lambda Functions.

To try your Lambda function

1. If you haven't already done so, do the following:

   a. Make sure the user using the client code has `lambda:InvokeFunction` permission. You can use the following permissions.

   ```
   {
       "Version": "2012-10-17",
       "Statement": [
           {
               "Sid": "LambdaPermission",
               "Effect": "Allow",
               "Action": "lambda:InvokeFunction",
               "Resource": "ARN for lambda function"
           }
       ]
   }
   ```

   You can get the ARN for your Lambda function function from the function overview in the Lambda console.

   To provide access, add permissions to your users, groups, or roles:

- Users and groups in AWS IAM Identity Center:

  Create a permission set. Follow the instructions in Create a permission set in the *AWS IAM Identity Center User Guide*.

- Users managed in IAM through an identity provider:

  Create a role for identity federation. Follow the instructions in Create a role for a third-party identity provider (federation) in the *IAM User Guide*.

- IAM users:

  - Create a role that your user can assume. Follow the instructions in Create a role for an IAM user in the *IAM User Guide*.
  - (Not recommended) Attach a policy directly to a user or add a user to a user group. Follow the instructions in Adding permissions to a user (console) in the *IAM User Guide*.

b. Install and configure AWS SDK for Python. For more information, see Step 4: Set up the AWS CLI and AWS SDKs.

c. Start the model that you specified in step 7 of Step 1: Create an AWS Lambda function (console) .

2. Save the following code to a file named `client.py`.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

"""
Purpose: Shows how to call the anomaly detection
AWS Lambda function.
"""
from botocore.exceptions import ClientError

import argparse
import logging
import base64
import json
import boto3
from os import environ

logger = logging.getLogger(__name__)
```

```
def analyze_image(function_name, image):
    """
    Analyzes an image with an AWS Lambda function.
    :param image: The image that you want to analyze.
    :return The status and classification result for
    the image analysis.
    """

    lambda_client = boto3.client('lambda')

    lambda_payload = {}

    if image.startswith('s3://'):
        logger.info("Analyzing image from S3 bucket: %s", image)
        bucket, key = image.replace("s3://", "").split("/", 1)
        s3_object = {
            'Bucket': bucket,
            'Name': key
        }
        lambda_payload = {"S3Object": s3_object}

    # Call the lambda function with the image.
    else:
        with open(image, 'rb') as image_file:
            logger.info("Analyzing local image image: %s ", image)
            image_bytes = image_file.read()
            data = base64.b64encode(image_bytes).decode("utf8")
            lambda_payload = {"image": data, "filename": image}

    response = lambda_client.invoke(FunctionName=function_name,
                                    Payload=json.dumps(lambda_payload))
    return json.loads(response['Payload'].read().decode())


def add_arguments(parser):
    """
    Adds command line arguments to the parser.
    :param parser: The command line parser.
    """

    parser.add_argument(
        "function", help="The name of the AWS Lambda function "
        "that you want to use to analyze the image.")
    parser.add_argument(
```

```
        "image", help="The local image that you want to analyze.")


def main():
    """
    Entrypoint for script.
    """
    try:
        logging.basicConfig(level=logging.INFO,
                            format="%(levelname)s: %(message)s")

        # Get command line arguments.
        parser = argparse.ArgumentParser(usage=argparse.SUPPRESS)
        add_arguments(parser)
        args = parser.parse_args()

        # Analyze image and display results.

        result = analyze_image(args.function, args.image)

        status = result['statusCode']

        if status == 200:
            classification = result['body']
            print(f"classification: {classification['Reject']}")
            print(f"Message: {classification['RejectMessage']}")
        else:
            print(f"Error: {result['statusCode']}")
            print(f"Message: {result['body']}")

    except ClientError as error:
        logging.error(error)
        print(error)


if __name__ == "__main__":
    main()
```

3. Run the code. For the command line argument, supply the Lambda function name and the path to a local image that you want to analyze. For example:

```
python client.py function_name /bucket/path/image.jpg
```

If successful, the output is a classification for the anomalies found in the image. If a classification isn't returned, consider lowering the confidence value that you set in step 7 of Step 1: Create an AWS Lambda function (console).

4. If you have finished with the Lambda function and the model isn't used by other applications, stop the model. Remember to start the model the next time you want use the Lambda function.

Using your Amazon Lookout for Vision model on an edge device

You can use your Amazon Lookout for Vision model on edge devices managed by AWS IoT Greengrass Version 2. AWS IoT Greengrass is an open source Internet of Things (IoT) edge runtime and cloud service. You can use it to build, deploy, and manage IoT applications on your devices. For more information, see AWS IoT Greengrass.

You deploy the same Amazon Lookout for Vision models that you've trained in the cloud onto AWS IoT Greengrass V2 compatible edge devices. You can then use your deployed model to perform anomaly detection on premises, such as a factory floor, without continually streaming data to the cloud. That way you can minimize bandwidth costs and detect anomalies locally with real-time image analysis.

> **Tip**
>
> Before deploying a Lookout for Vision model with AWS IoT Greengrass, we recommend that you read the *AWS IoT Greengrass Version 2 developer guide*. For more information, see What is AWS IoT Greengrass?.

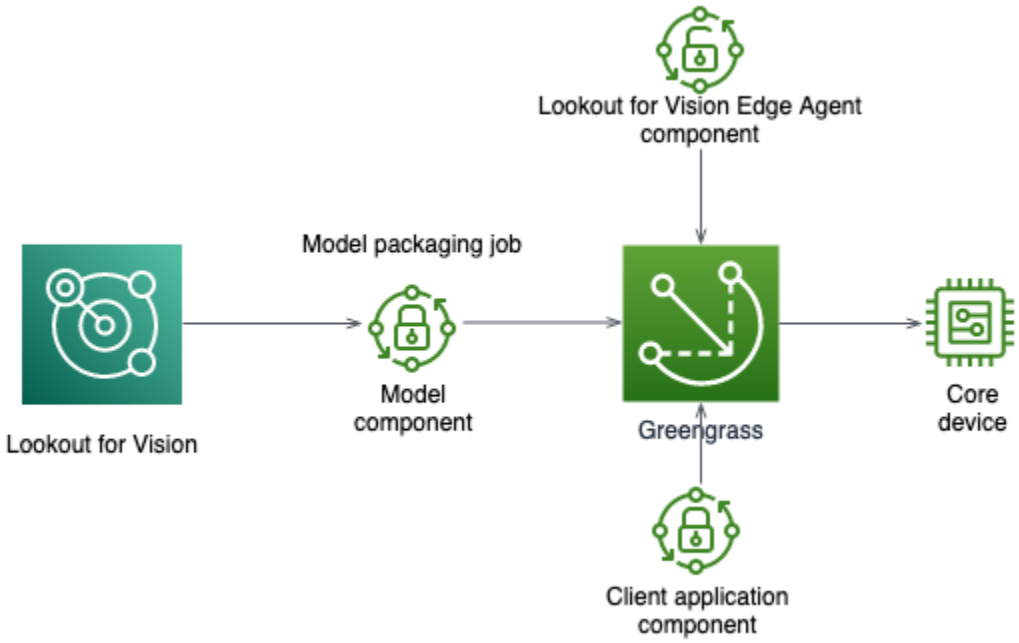
To use a Lookout for Vision model on an AWS IoT Greengrass V2 core device, you deploy the model and supporting software as components to the core device. A component is a software module, such as a Lookout for Vision model, that runs on a Greengrass core device. There are two forms of component. A custom component is a component that you create and is only accessible to you. It is also known as a private component. An AWS supplied component is a pre-built component that AWS provides. It is also known as a public component. For more information, see https://docs.aws.amazon.com/greengrass/v2/developerguide/public-components.html.

The components that you deploy to a core device for a Lookout for Vision model and supporting software are:

- *Model component*. A custom component that contains your Lookout for Vision model. To create the model component, you use Lookout for Vision to create a *model packaging job*. A model packaging job creates a component for the model and makes it available as a custom component within AWS IoT Greengrass V2. For more information, see Packaging your Amazon Lookout for Vision model.

- *Client application component*. A custom component you create that implements the code for your business requirements. For example, finding anomalous circuit boards from images taken after assembly. For more information, see Writing your client application component.
- *Amazon Lookout for Vision Edge Agent component*. An AWS supplied component that provides an API for using and managing your model. For example, code in your client application component can use the `DetectAnomalies` API to detect anomalies in images. The Lookout for Vision Edge Agent component is a dependency of the model component. It is automatically installed on the core device when you deploy the model component. For more information, see Amazon Lookout for Vision Edge Agent API reference.

After you create the model component and client application component, you can use AWS IoT Greengrass V2 to deploy the components and dependencies to the core device. For more information, see Deploying your components to a device.



Important

The predictions that your model makes with `DetectAnomalies` on a core device might differ from predictions made using the same model hosted in the cloud. We recommend that you test your model on a core device before using it in a production environment. To reduce prediction mismatches between device hosted models and cloud hosted models, we recommend increasing the number of normal and anomalous images in your training

dataset. We don't recommend reusing existing images to increase the size of the training dataset.

Deploying a model and client application component to a AWS IoT Greengrass Version 2 core device

The procedure for deploying an Amazon Lookout for Vision model and client application component on an AWS IoT Greengrass Version 2 core device is as follows:

1. Set up your core devices with AWS IoT Greengrass Version 2.
2. Create a model packaging job by using Lookout for Vision. The job creates your model component.
3. Write a client application component. The component implements your business logic.
4. Deploy the model component and client application component to the core device by using AWS IoT Greengrass V2.

After the components and dependencies are deployed to the core device, you can use the model on the core device.

> **Note**
>
> You must use the same AWS Region and AWS account to create and deploy your Lookout for Vision model and client application component.

AWS IoT Greengrass Version 2 core device requirements

To use an Amazon Lookout for Vision model on an AWS IoT Greengrass Version 2 core device, your model has various requirements of the core device.

Topics

- Tested devices, chip architectures, and operating systems
- Core device memory and storage
- Required software

Tested devices, chip architectures, and operating systems

We expect Amazon Lookout for Vision to work on the following hardware:

- CPU architectures
  - X86_64 (64-bit version of the x86 instruction set)
  - Aarch64 (ARMv8 64-bit CPU)
- (GPU accelerated inference only) NVIDIA GPU Accelerator with sufficient memory capacity (At least 6.0 GB for a running model).

The Amazon Lookout for Vision team has tested Lookout for Vision models on the following devices, chip architectures, and operating systems.

Devices

Device	Operating system	Architecture	Accelerator	Compiler options
jetson_xavier (NVIDIA® Jetson AGX Xavier)	Linux	Aarch64	NVIDIA	{"gpu-code": "sm_72", "trt-ver": "7.1.3", "cuda-ver": "10.2"} {"gpu-code": "sm_72", "trt-ver": "8.2.1", "cuda-ver": "10.2"}
g4dn.xlarge (EC2 Instances (G4) with NVIDIA	Linux	X86_64/X86-64	NVIDIA	{"gpu-code":

Device	Operating system	Architecture	Accelerator	Compiler options
T4 Tensor Core GPUs)				`"sm_75", "trt-ver" : "7.1.3", "cuda-ver": "10.2"}`
g5.xlarge (EC2 Instances (G5) with NVIDIA A10G Tensor Core GPUs)	Linux	X86_64/X86-64	NVIDIA	`{"gpu-code": "sm_80", "trt-ver" : "8.2.0", "cuda-ver": "11.2"}`
c5.2xlarge (Amazon EC2 C5 Instances)	Linux	X86_64/X86-64	CPU	`{"mcpu": "core-avx2"}`

Core device memory and storage

To run a single model and the Amazon Lookout for Vision Edge Agent, your core device has the following memory and storage requirements. You might need more memory and storage for your client application component.

- Storage – At least 1.5 GB.
- Memory – At least 6.0 GB for a running model.

Required software

A core device requires the following software.

Jetson Devices

If your core device is a Jetson device, you need the following software installed on the core device.

Software	Supported versions
Jetpack SDK	4.4 to 4.6.1
Python and Python virtual environment for Lookout for Vision Edge Agent version 1.x	3.8 or 3.9

X86 hardware

If your core device uses x86 hardware, you need the following software installed on the core device.

CPU inference

Software	Supported versions
Python and Python virtual environment for Lookout for Vision Edge Agent version 1.x	3.8 or 3.9

GPU accelerated inference

Software versions vary depending on the microarchitecture of the NVIDIA GPU that you use.

NVIDIA GPU with microarchitecture prior to Ampere (compute capability is less than 8.0)

Required software for an NVIDIA GPU with a microarchitecture prior to Ampere (compute capability that is less than 8.0). The `gpu-code` must be less than `sm_80`.

Software	Supported versions
NVIDIA CUDA	10.2
NVIDIA TensorRT	At least 7.1.3 and less than 8.0.0
Python and Python virtual environment for Lookout for Vision Edge Agent version 1.x	3.8 or 3.9

NVIDIA GPU with Ampere microarchitecture (compute capability 8.0)

Required software for an NVIDIA GPU with the Ampere microarchitecture (compute capability is 8.0). The `gpu-code` must be `sm_80`).

Software	Supported versions
NVIDIA CUDA	11.2
NVIDIA TensorRT	8.2.0
Python and Python virtual environment for Lookout for Vision Edge Agent version 1.x	3.8 or 3.9

Setting up your AWS IoT Greengrass Version 2 core device

Amazon Lookout for Vision uses AWS IoT Greengrass Version 2 to simplify the deployment of the model component, Amazon Lookout for Vision Edge Agent component, and client application component to your AWS IoT Greengrass V2 core device. For information about the devices and hardware that you can use, see AWS IoT Greengrass Version 2 core device requirements.

Setting up your core device

Use the following information to set up your core device.

To set up your core device

1. Set up your GPU libraries. Don't do this step if you aren't using GPU accelerated inference.
   a. Verify that you have a GPU that supports CUDA. For more information, see Verify You Have a CUDA-Capable GPU.
   b. Setup CUDA, cuDNN, and TensorRT on your device by doing one of the following:
      - If you are using a Jetson device, install JetPack version 4.4 - 4.6.1. For more information, see JetPack Archive.
      - If you are using x86 based hardware, and your NVIDIA GPU microarchitecture is prior to Ampere (compute capability is less than 8.0), do the following:

1. Set up CUDA version 10.2 by following the instructions at NVIDIA CUDA Installation Guide for Linux.
2. Install cuDNN, by following the instructions at NVIDIA cuDNN Documentation.
3. Set up TensorRT (version 7.1.3 or later, but earlier than 8.0.0) by following the instructions at NVIDIA TENSORRT DOCUMENTATION.

- If you are using x86 based hardware, and your NVIDIA GPU microarchitecture is Ampere (compute capability is 8.0), do the following:
  1. Set up CUDA (version 11.2) by following the instructions at NVIDIA CUDA Installation Guide for Linux.
  2. Install cuDNN, by following the instructions at NVIDIA cuDNN Documentation.
  3. Set up TensorRT (version 8.2.0) by following the instructions at NVIDIA TENSORRT DOCUMENTATION.

2. Install the AWS IoT Greengrass Version 2 core software on your core device. For more information, see Install the AWS IoT Greengrass Core software in the *AWS IoT Greengrass Version 2 Developer Guide*.
3. To read from the Amazon S3 bucket that stores the model, attach permission to the IAM role (token exchange role) that you create during AWS IoT Greengrass Version 2 setup. For more information, see Allow access to S3 buckets for component artifacts.
4. At the command prompt, enter the folllowing command to install Python and a Python virtual environment onto the core device.

```
sudo apt install python3.8 python3-venv python3.8-venv
```

5. Use the following command to add the Greengrass user to the video group. This lets Greengrass deployed components access the GPU:

```
sudo usermod -a -G video ggc_user
```

6. (Optional) If you want to call Lookout for Vision Edge Agent API from a different user, add the required user to the `ggc_group`. This lets the user communicate with the Lookout for Vision Edge Agent over the Unix Domain socket:

```
sudo usermod -a -G ggc_group $(whoami)
```

Packaging your Amazon Lookout for Vision model

A model packaging job packages an Amazon Lookout for Vision model as a model component.

To create a model packaging job, you choose the model you want to package and provide settings for the model component that the job creates. You can only package a model that has been successfully trained.

You can use the Lookout for Vision console or AWS SDK to create the model packaging job. You can also get information about the model packaging jobs you create. For more information, see Getting information about model packaging jobs. You can use AWS IoT Greengrass V2 console or the AWS SDK to deploy the components to the AWS IoT Greengrass Version 2 core device.

Topics

- Package settings
- Packaging your model (Console)
- Packaging your model (SDK)
- Getting information about model packaging jobs

Package settings

Use the following information to decide the package settings for your model packaging job.

To create a model packaging job, see Packaging your model (Console) or Packaging your model (SDK).

Topics

- Target hardware
- Component settings

Target hardware

You can choose a target device or target platform for your model, but not both. For more information, see Tested devices, chip architectures, and operating systems.

Target device

The target device for the model, such as NVIDIA® Jetson AGX Xavier. You don't need to specify compiler options.

Target platform

Amazon Lookout for Vision supports the following platform configurations:

- X86_64 (64-bit version of the x86 instruction set) and Aarch64 (ARMv8 64-bit CPU) architectures.
- Linux operating system.
- Inference using NVIDIA or CPU accelerators.

You need to specify the correct compiler options for your target platform.

Compiler options

Compiler options allow you to specify the target platform for your AWS IoT Greengrass Version 2 core device. Currently you can specify the following compiler options.

NVIDIA accelerator

- `gpu-code` — Specifies the gpu code of the core device that runs the model component.
- `trt-ver` — Specifies the TensorRT version in x.y.z. format.
- `cuda-ver` — Specifies the CUDA version in x.y format.

CPU accelerator

- (Optional) `mcpu` — specifies the instruction set. For example `core-avx2`. If you don't provide a value, Lookout for Vision uses the value `core-avx2`.

You specify the options in JSON format. For example:

```
{"gpu-code": "sm_75", "trt-ver": "7.1.3", "cuda-ver": "10.2"}
```

For more examples, see Tested devices, chip architectures, and operating systems.

Component settings

The model packaging job creates a model component that contains your model. The job creates artifacts that AWS IoT Greengrass V2 uses to deploy the model component to the core device.

You can't create a model component with the same component name and component version as an existing component.

Component name

A name for the model component that Lookout for Vision creates during model packaging. The component name you specify is displayed in the AWS IoT Greengrass V2 console. You use the component name in the recipe that you create for the client application component. For more information, see Creating the client application component.

Component description

(Optional) A description for the model component.

Component version

A version number for the model component. You can accept the default version number or choose your own. The version number must follow the semantic version number system – major.minor.patch. For example, version 1.0.0 represents the first major release for a component. For more information, see Semantic Versioning 2.0.0. If you don't provide a value, Lookout for Vision uses the version number of your model to generate a version for you.

Component location

The Amazon S3 location where you want the model packaging job to save the model component artifacts. The Amazon S3 bucket must be in the same AWS Region and AWS account in which you use AWS IoT Greengrass Version 2. To create an Amazon S3 bucket, see Creating a bucket.

Tags

You can identify, organize, search for, and filter your components by using tags. Each tag is a label consisting of a user-defined key and value. The tags are attached to the model component when the model packaging job creates the model component in Greengrass. A component is an AWS IoT Greengrass V2 resource. The tags aren't attached to any of your Lookout for Vision resources, such as your models. For more information, see Tagging AWS resources.

Packaging your model (Console)

You can create a model packaging job by using the Amazon Lookout for Vision console.

For information about package settings, see Package settings.

To package a model (console)

1. Create an Amazon S3 bucket, or reuse an existing bucket, that Lookout for Vision uses to store the packaging job artifacts (model component).
2. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
3. Choose **Get started**.
4. In the left navigation pane, choose **Projects**.
5. In the **Projects** section, choose the project that contains the model you want to package.
6. In the left navigation pane, under the project name, choose **Edge model packages**.
7. In the **Model packaging jobs** section, choose **Create model packaging job**.
8. Enter the settings for the package. For more information, see Package settings.
9. Choose **Create model packaging job**.
10. Wait until the packaging job finishes. The job is finished when the status of the job is **Success**.
11. Choose the packaging job in the **Model packaging jobs** section.
12. Choose **Continue deployment in Greengrass** to continue deployment of your model component in AWS IoT Greengrass Version 2. For more information, see Deploying your components to a device.

Packaging your model (SDK)

You package a model as a model component by creating a model packaging job. To create a model packaging job you call the StartModelPackagingJob API. The job might take a while to complete. To find out the current status, call DescribeModelPackagingJob and check the `Status` field in the response.

For information about package settings, see Package settings.

The following procedure shows you how to start a packaging job by using the AWS CLI. You can package the model for a target platform or a target device. For example Java code, see StartModelPackagingJob.

To package your model (SDK)

1. If you haven't already done so, install and configure the AWS CLI and the AWS SDKs. For more information, see Step 4: Set up the AWS CLI and AWS SDKs.
2. Make sure that you have the correct permissions to start a model packaging job. For more information, see StartModelPackagingJob.
3. Use the following CLI commands to package your model for either a target device or a target platform.

   Target platform

   The following CLI command shows how to package a model for a target platform with an NVIDIA accelerator.

   Change the following values:

   - `project_name` to the name of the project that contains the model that you want to package.
   - `model_version` to the version of the model that you want to package.
   - (Optional) `description` to a description for your model packaging job.
   - `architecture` to the architecture (`ARM64` or `X86_64`) of the AWS IoT Greengrass Version 2 core device where you run the model component.
   - `gpu_code` to the gpu code of the core device where you run the model component.
   - `trt_ver` to the TensorRT version you have installed on your core device.
   - `cuda_ver` to the CUDA version you have installed on your core device.
   - `component_name` to a name for the model component that you want to create on AWS IoT Greengrass V2.
   - (Optional) `component_version` to a version for the model component that the packaging job creates. Use the format `major.minor.patch`. For example, 1.0.0 represents the first major release for a component.
   - `bucket` to the Amazon S3 bucket where the packaging job stores the model component artifacts.

- `prefix` to the location within the Amazon S3 bucket where the packaging job stores the model component artifacts.
- (Optional) `component_description` to a description for the model component.
- (Optional) `tag_key1` and `tag_key2` to the keys for tags that are attached to the model component.
- (Optional) `tag_value1` and `tag_value2` to the key values for the tags that are attached to the model component.

```
aws lookoutvision start-model-packaging-job \
  --project-name project_name \
  --model-version model_version \
  --description="description" \
  --configuration
 "Greengrass={TargetPlatform={Os='LINUX',Arch='architecture',Accelerator='NVIDIA'},Compi
code\": \"gpu_code\", \"trt-ver\": \"trt_ver\", \"cuda-ver\":
 \"cuda_ver\"}',S3OutputLocation={Bucket='bucket',Prefix='prefix'},ComponentName='Compon
 {Key='tag_key2',Value='tag_value2'}]}" \
  --profile lookoutvision-access
```

For example:

```
aws lookoutvision start-model-packaging-job \
  --project-name test-project-01 \
  --model-version 1 \
  --description="Model Packaging Job for G4 Instance using TargetPlatform
 Option" \
  --configuration
 "Greengrass={TargetPlatform={Os='LINUX',Arch='X86_64',Accelerator='NVIDIA'},CompilerOpt
code\": \"sm_75\", \"trt-ver\": \"7.1.3\", \"cuda-ver\":
 \"10.2\"}',S3OutputLocation={Bucket='bucket',Prefix='test-project-01/
folder'},ComponentName='SampleComponentNameX86TargetPlatform',ComponentVersion='0.1.0',C
 is my component',Tags=[{Key='modelKey0',Value='modelValue'},
 {Key='modelKey1',Value='modelValue'}]}" \
  --profile lookoutvision-access
```

Target Device

Use the following CLI commands to package a model for a target device.

Change the following values:

- `project_name` to the name of the project that contains the model that you want to package.
- `model_version` to the version of the model that you want to package.
- (Optional) `description` to a description for your model packaging job.
- `component_name` to a name for the model component that you want to create on AWS IoT Greengrass V2.
- (Optional) `component_version` to a version for the model component that the packaging job creates. Use the format `major.minor.patch`. For example, 1.0.0 represents the first major release for a component.
- `bucket` to the Amazon S3 bucket where the packaging job stores the model component artifacts.
- `prefix` to the location within the Amazon S3 bucket where the packaging job stores the model component artifacts.
- (Optional) `component_description` to a description for the model component.
- (Optional) `tag_key1` and `tag_key2` to the keys for tags that are attached to the model component.
- (Optional) `tag_value1` and `tag_value2` to the key values for the tags that are attached to the model component.

```
aws lookoutvision start-model-packaging-job \
  --project-name project_name \
  --model-version model_version \
  --description="description" \
  --configuration
 "Greengrass={TargetDevice='jetson_xavier',S3OutputLocation={Bucket='bucket',Prefix='pre
 {Key='tag_key2',Value='tag_value2'}]}" \
  --profile lookoutvision-access
```

For example:

```
aws lookoutvision start-model-packaging-job \
  --project-name project_01 \
```

```
 --model-version 1 \
 --description="description" \
 --configuration
"Greengrass={TargetDevice='jetson_xavier',S3OutputLocation={Bucket='bucket',Prefix='com
model component',Tags=[{Key='tag_key1',Value='tag_value1'},
{Key='tag_key2',Value='tag_value2'}]}" \
 --profile lookoutvision-access
```

4. Note the value of JobName in the response. You need it in the next step. For example:

```
{
    "JobName": "6bcfd0ff-90c3-4463-9a89-6b4be3daf972"
}
```

5. Use `DescribeModelPackagingJob` to get the current status of the job. Change the following:

   - `project_name` to the name of the project that you are using.
   - `job_name` to the name of the job that you noted in the previous step.

```
aws lookoutvision describe-model-packaging-job \
    --project-name project_name \
    --job-name job_name \
    --profile lookoutvision-access
```

   The model packaging job is complete if the value of `Status` is SUCCEEDED. If the value is different, wait a minute and try again.

6. Continue deployment using AWS IoT Greengrass V2. For more information, see Deploying your components to a device.

Getting information about model packaging jobs

You can use the Amazon Lookout for Vision console and AWS SDK to get information about the model packaging jobs that you create.

Topics

- Getting model packaging job information (Console)
- Getting model packaging job information (SDK)

Getting model packaging job information (Console)

To get model packaging job information (console)

1. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
2. Choose **Get started**.
3. In the left navigation pane, choose **Projects**.
4. In the **Projects** section, choose the project that contains the model packaging job you want to view.
5. In the left navigation pane, under the project name, choose **Edge model packages**.
6. In the **Model packaging job** section, choose the model packaging job that you want to view. The details page for the model packaging job is shown.

Getting model packaging job information (SDK)

You can use the AWS SDK to list the model packaging jobs in a project and get information about a specific model packaging job.

List model packaging jobs

You can list the model packaging jobs in a project by calling the ListModelPackagingJobs API. The response includes a list of ModelPackagingJobMetadata objects that provides information about each model packaging job. Also included is a pagination token that you can use to get the next set of results, if the list is incomplete.

To list your model packaging jobs

1. If you haven't already done so, install and configure the AWS CLI and the AWS SDKs. For more information, see Step 4: Set up the AWS CLI and AWS SDKs.
2. Use the following CLI command. Change `project_name` to the name of the project that you want to use.

```
aws lookoutvision list-model-packaging-jobs \
  --project-name project_name \
```

```
  --profile lookoutvision-access
```

Describe a model packaging job

Use the DescribeModelPackagingJob API to get information about a model packaging job. The response is a ModelPackagingDescription object that includes the current status of the job and other information.

To describe a package

1. If you haven't already done so, install and configure the AWS CLI and the AWS SDKs. For more information, see Step 4: Set up the AWS CLI and AWS SDKs.
2. Use the following CLI command. Change the following:

   - `project_name` to the name of the project that you are using.
   - `job_name` to the name of the job. You get the job name (`JobName`) when you call StartModelPackagingJob.

```
aws lookoutvision describe-model-packaging-job \
    --project-name project_name \
    --job-name job_name \
    --profile lookoutvision-access
```

Writing your client application component

A client application component is a custom AWS IoT Greengrass Version 2 component that you write. It implements the business logic you need to use an Amazon Lookout for Vision model on an AWS IoT Greengrass Version 2 core device.

To access a model, your client application component uses the Lookout for Vision Edge Agent component. The Lookout for Vision Edge Agent component provides an API that you use to analyze images with a model and manage the models on a core device.

The Lookout for Vision Edge Agent API is implemented using gRPC, which is a protocol for making remote procedure calls. For more information, see gRPC. To write your code, you can use any language supported by gRPC. We provide example Python code. For more information, see Using a model in your client application component.

> **Note**
>
> The Lookout for Vision Edge Agent component is a dependency of the model component that you deploy. It is automatically deployed to the core device when you deploy the model component to the core device.

To write a client application component, you do the following.

1. Set up your environment to use gRPC and install third-party libraries.
2. Write code to use the model.
3. Deploy the code as a custom component to the core device.

For an example client application component that shows how to perform anomaly detection in a custom GStreamer pipeline, see https://github.com/awslabs/aws-greengrass-labs-lookoutvision-gstreamer.

Setting up your environment

To write client code, your development environment connects remotely to an AWS IoT Greengrass Version 2 core device to which you have deployed a Amazon Lookout for Vision model component and dependencies. Alternatively, you can write code on a core device. For more information, see AWS IoT Greengrass development tools and Develop AWS IoT Greengrass components.

Your client code should use gRPC client to access the Amazon Lookout for Vision Edge Agent. This section shows how to set up your development environment with gRPC and install third-party dependencies needed for the `DetectAnomalies` example code.

After you finish writing your client code, you create a custom component and deploy the custom component to your edge devices. For more information, see Creating the client application component.

Topics

- Setting up gRPC
- Adding third-party dependencies

Setting up gRPC

In your development environment, you need a gRPC client that you use in your code to call the Lookout for Vision Edge Agent API. To do this, you create a gRPC stub by using a `.proto`service definition file for the Lookout for Vision Edge Agent.

> **Note**
>
> You can also get the service definition file from the Lookout for Vision Edge Agent application bundle. The application bundle is installed when the Lookout for Vision Edge Agent component is installed as a dependency of the model component. The application bundle is located at `/greengrass/v2/packages/artifacts-unarchived/aws.iot.lookoutvision.EdgeAgent/`*`edge_agent_version`*`/lookoutvision_edge_agent`. Replace `edge_agent_version` with version of the Lookout for Vision Edge Agent that you are using. To get the application bundle, you need to deploy the Lookout for Vision Edge Agent to a core device.

To set up gRPC

1. Download the zip file, proto.zip. The zip file contains the .proto service definition file (`edge-agent.proto`).
2. Unzip the content.
3. Open a command prompt and navigate to the folder that contains `edge-agent.proto`.
4. Use the following commands to generate the Python client interfaces.

   ```
   %%bash
   python3 -m pip install grpcio
   python3 -m pip install grpcio-tools
   python3 -m grpc_tools.protoc --proto_path=. --python_out=. --grpc_python_out=.
    edge-agent.proto
   ```

   If the commands are successful, the stubs `edge_agent_pb2_grpc.py` and `edge_agent_pb2.py` are created in the working directory.
5. Write the client code that uses your model. For more information, see Using a model in your client application component.

Adding third-party dependencies

The `DetectAnomalies` example code uses the Pillow library to work with images. For more information, see Detecting Anomalies by using image bytes.

Use the following command to install the Pillow library.

```
python3 -m pip install Pillow
```

Using a model in your client application component

The steps for using a model from a client application component are similar to using a model hosted in the cloud.

1. Start running the model.
2. Detect anomalies in images.
3. Stop the model, if no longer needed.

The Amazon Lookout for Vision Edge Agent provides API to start a model, detect anomalies in an image, and stop a model. You can also use the API to list the models on a device and get information about a deployed model. For more information, see Amazon Lookout for Vision Edge Agent API reference.

You can get error information by checking the gRPC status codes. For more information, see Getting error information.

To write your code, you can use any language supported by gRPC. We provide example Python code.

Topics

- Using the stub in your client application component
- Starting the model
- Detecting anomalies
- Stopping the model
- Listing models on a device
- Describing a model
- Getting error information

Using the stub in your client application component

Use the following code to set up access to your model through the Lookout for Vision Edge Agent.

```
import grpc
from edge_agent_pb2_grpc import EdgeAgentStub
import edge_agent_pb2 as pb2

# Creating stub.
 with grpc.insecure_channel("unix:///tmp/aws.iot.lookoutvision.EdgeAgent.sock") as
 channel:
    stub = EdgeAgentStub(channel)
    # Add additional code that works with Edge Agent in this block to prevent resources
 leakage
```

Starting the model

You start a model by calling the StartModel API. The model might take a while to start. You can check the current status by calling DescribeModel. The model is running if the value of the `status` field is Running.

Example code

Replace *`component_name`* with the name of your model component.

```
import time

import grpc
from edge_agent_pb2_grpc import EdgeAgentStub
import edge_agent_pb2 as pb2


model_component_name = "component_name"


def start_model_if_needed(stub, model_name):
    # Starting model if needed.
    while True:
        model_description_response =
 stub.DescribeModel(pb2.DescribeModelRequest(model_component=model_name))
        print(f"DescribeModel() returned {model_description_response}")
        if model_description_response.model_description.status == pb2.RUNNING:
            print("Model is already running.")
```

```
            break
        elif model_description_response.model_description.status == pb2.STOPPED:
            print("Starting the model.")
            stub.StartModel(pb2.StartModelRequest(model_component=model_name))
            continue
        elif model_description_response.model_description.status == pb2.FAILED:
            raise Exception(f"model {model_name} failed to start")
        print(f"Waiting for model to start.")
        if model_description_response.model_description.status != pb2.STARTING:
            break
        time.sleep(1.0)


# Creating stub.
with grpc.insecure_channel("unix:///tmp/aws.iot.lookoutvision.EdgeAgent.sock") as
 channel:
    stub = EdgeAgentStub(channel)
    start_model_if_needed(stub, model_component_name)
```

Detecting anomalies

You use the DetectAnomalies API to detect anomalies in an image.

The `DetectAnomalies` operation expects the image bitmap to be passed in RGB888 packed format. The first byte represents the red channel, the second byte represents the green channel, and the third byte represents the blue channel. If you provide the image in a different format, such as BGR, the predictions from DetectAnomalies are incorrect.

By default, OpenCV uses the BGR format for image bitmaps. If you are using OpenCV to capture images for analysis with `DetectAnomalies`, you must convert the image to RGB888 format before you pass the image to `DetectAnomalies`.

The images you supply to `DetectAnomalies` must have the same width and height dimensions as the images that you used to train the model.

Detecting Anomalies by using image bytes

You can detect anomalies in an image by supplying the image as image bytes. In the following example, the image bytes are retrieved from an image stored in the local file system.

Replace *`sample.jpg`* with the name of the image file that you want to analyze. Replace *`component_name`* with the name of your model component.

```
import time

from PIL import Image
import grpc
from edge_agent_pb2_grpc import EdgeAgentStub
import edge_agent_pb2 as pb2


model_component_name = "component_name"

....
# Detecting anomalies.
def detect_anomalies(stub, model_name, image_path):
    image = Image.open(image_path)
    image = image.convert("RGB")
    detect_anomalies_response = stub.DetectAnomalies(
        pb2.DetectAnomaliesRequest(
            model_component=model_name,
            bitmap=pb2.Bitmap(
                width=image.size[0],
                height=image.size[1],
                byte_data=bytes(image.tobytes())
            )
        )
    )
    print(f"Image is anomalous -
 {detect_anomalies_response.detect_anomaly_result.is_anomalous}")
    return detect_anomalies_response.detect_anomaly_result


# Creating stub.
with grpc.insecure_channel("unix:///tmp/aws.iot.lookoutvision.EdgeAgent.sock") as
 channel:
    stub = EdgeAgentStub(channel)
    start_model_if_needed(stub, model_component_name)
    detect_anomalies(stub, model_component_name, "sample.jpg")
```

Detecting Anomalies by using shared memory segment

You can detect anomalies in an image by supplying the image as image bytes in the POSIX shared memory segment. For best performance, we recommend using shared memory for DetectAnomalies requests. For more information, see DetectAnomalies.

Stopping the model

If you are no longer using the model, the StopModel API to stop the model running.

```
stop_model_response = stub.StopModel(
    pb2.StopModelRequest(
         model_component=model_component_name
    )
)
print(f"New status of the model is {stop_model_response.status}")
```

Listing models on a device

You can use the the section called “ListModels” API to list the models that are deployed to a device.

```
models_list_response = stub.ListModels(
    pb2.ListModelsRequest()
)
for model in models_list_response.models:
    print(f"Model Details {model}")
```

Describing a model

You can get information about a model that's deployed to a device by calling the DescribeModel API. Using `DescribeModel` is useful for getting the current status of a model. For example, you need to know if a model is running before you can call `DetectAnomalies`. For example code, see Starting the model.

Getting error information

gRPC status codes are used to report API results.

You can get error information by catching the `RpcError` exception, as shown in the following example. For information about the error status codes, see the reference topic for an API.

```
# Error handling.
try:
    stub.DetectAnomalies(detect_anomalies_request)
except grpc.RpcError as e:
```

```
    print(f"Error code: {e.code()}, Status: {e.details()}")
```

Creating the client application component

You can create the client application component once you have generated your gRPC stubs and you have your client application code ready. The component you create is a custom component that you deploy to an AWS IoT Greengrass Version 2 core device with AWS IoT Greengrass V2. A recipe that you create describes your custom component. The recipe includes any dependencies that also need to be deployed. In this case, you specify the model component that you create in Packaging your Amazon Lookout for Vision model. For more information about component recipes, see AWS IoT Greengrass Version 2 component recipe reference.

The procedures on this topic show you how to create the client application component from a recipe file and publish it as an AWS IoT Greengrass V2 custom component. You can use the AWS IoT Greengrass V2 console or the AWS SDK to publish the component.

For detailed information about creating a custom component, see the following in the *AWS IoT Greengrass V2* documentation.

- Develop and test a component on your device
- Create AWS IoT Greengrass components
- Publish components to deploy to your core devices

Topics

- IAM permissions for publishing a client application component
- Creating the recipe
- Publishing the client application component (Console)
- Publishing the client application component (SDK)

IAM permissions for publishing a client application component

To create and publish your client application component, you need the following IAM permissions:

- `greengrass:CreateComponentVersion`
- `greengrass:DescribeComponent`
- `s3:PutObject`

Creating the recipe

In this procedure, you create the recipe for a simple client application component. The code in `lookoutvision_edge_agent_example.py` lists the models that are deployed to the device and is automatically run after you deploy the component to the core device. To view the output, check the component log after you deploy the component. For more information, see Deploying your components to a device. When you are ready, use this procedure to create the recipe for code that implements your business logic.

You create the recipe as a JSON or YAML format file. You also upload the client application code to an Amazon S3 bucket.

To create the client application component recipe

1. If you haven't already, create the gRPC stub files. For more information, see Setting up gRPC.
2. Save the following code to a file named `lookoutvision_edge_agent_example.py`

```
import grpc
from edge_agent_pb2_grpc import EdgeAgentStub
import edge_agent_pb2 as pb2

# Creating stub.
with grpc.insecure_channel("unix:///tmp/aws.iot.lookoutvision.EdgeAgent.sock") as
 channel:
    stub = EdgeAgentStub(channel)
    # Add additional code that works with Edge Agent in this block to prevent
 resources leakage

    models_list_response = stub.ListModels(
        pb2.ListModelsRequest()
    )
    for model in models_list_response.models:
        print(f"Model Details {model}")
```

3. Create an Amazon S3 bucket (or use an existing bucket) to store the source files for your client application component. The bucket must in your AWS account and in the same AWS Region in which you use AWS IoT Greengrass Version 2 and Amazon Lookout for Vision.
4. Upload `lookoutvision_edge_agent_example.py`, `edge_agent_pb2_grpc.py` and `edge_agent_pb2.py` to the Amazon S3 bucket that you created in the previous step.

Note the Amazon S3 path of each file. You created `edge_agent_pb2_grpc.py` and `edge_agent_pb2.py` in Setting up gRPC.

5. In an editor create the following JSON or YAML recipe file.

   - `model_component` to the name of your model component. For more information, see Component settings.
   - Change the URI entries to the S3 paths of `lookoutvision_edge_agent_example.py`, `edge_agent_pb2_grpc.py`, and `edge_agent_pb2.py`.

   JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.lookoutvision.EdgeAgentPythonExample",
  "ComponentVersion": "1.0.0",
  "ComponentType": "aws.greengrass.generic",
  "ComponentDescription": "Lookout for Vision Edge Agent Sample Application",
  "ComponentPublisher": "Sample App Publisher",
  "ComponentDependencies": {
    "model_component": {
      "VersionRequirement": ">=1.0.0",
      "DependencyType": "HARD"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "install": "pip3 install grpcio grpcio-tools protobuf Pillow",
        "run": {
          "script": "python3 {artifacts:path}/
lookoutvision_edge_agent_example.py"
        }
      },
      "Artifacts": [
        {
          "Uri": "S3 path to lookoutvision_edge_agent_example.py"
        },
        {
```

```
          "Uri": "S3 path to edge_agent_pb2_grpc.py"
        },
        {
          "Uri": "S3 path to edge_agent_pb2.py"
        }
      ]
    }
  ],
  "Lifecycle": {}
}
```

YAML

```
---
RecipeFormatVersion: 2020-01-25
ComponentName: com.lookoutvison.EdgeAgentPythonExample
ComponentVersion: 1.0.0
ComponentDescription: Lookout for Vision Edge Agent Sample Application
ComponentPublisher: Sample App Publisher
ComponentDependencies:
  model_component:
    VersionRequirement: '>=1.0.0'
    DependencyType: HARD
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      install: |-
        pip3 install grpcio
        pip3 install grpcio-tools
        pip3 install protobuf
        pip3 install Pillow
      run:
        script: |-
          python3 {artifacts:path}/lookout_vision_agent_example.py
    Artifacts:
      - URI: S3 path to lookoutvision_edge_agent_example.py
      - URI: S3 path to edge_agent_pb2_grpc.py
      - URI: S3 path to edge_agent_pb2.py
```

6. Save the JSON or YAML file to your computer.
7. Create the client application component by doing one of the following:

- If you want to use the AWS IoT Greengrass console, do Publishing the client application component (Console).
- If you want use the AWS SDK, do Publishing the client application component (SDK).

Publishing the client application component (Console)

You can use the AWS IoT Greengrass V2 console to publish the client application component.

To publish the client application component

1. If you haven't already, create the recipe for your client appication component by doing Creating the recipe.
2. Open the AWS IoT Greengrass console at https://console.aws.amazon.com/iot/
3. In the left navigation pane, under **Greengrass** choose **Components**.
4. Under **My components** choose **Create component**.
5. On the **Create component page** choose **Enter recipe as JSON** if you want to use a JSON format recipe. Choose **Enter recipe as YAML** if you want to use a YAML format recipe.
6. Under **Recipe** replace the existing recipe with the JSON or YAML recipe that you created in Creating the recipe.
7. Choose **Create component**.
8. Next, deploy your client application component.

Publishing the client application component (SDK)

You can publish the client application component by using the CreateComponentVersion API.

To publish the client application component (SDK)

1. If you haven't already, create the recipe for your client appication component by doing Creating the recipe.
2. At the command prompt, enter the following command to create the client application component. Replace `recipe-file` with the name of the recipe file you created in Creating the recipe.

```
aws greengrassv2 create-component-version --inline-recipe fileb://recipe-file
```

Note the ARN of the component in the response. You need it in the next step.

3. Use the following command to get the status of the client application component. Replace `component-arn` with the ARN that you noted in the previous step. The client application component is ready if the value of `componentState` is `DEPLOYABLE`.

```
aws greengrassv2 describe-component --arn component-arn
```

4. Next, deploy your client application component.

Deploying your components to a device

To deploy the model component and client application component to an AWS IoT Greengrass Version 2 core device you use the AWS IoT Greengrass V2 console or use the CreateDeployment API. For more information, see Create deployments or in the *AWS IoT Greengrass Version 2 Developer Guide*. For information about updating a component that is deployed to a core device, see Revise deployments.

Topics

- IAM permissions for deploying components
- Deploying your components (console)
- Deploying the components (SDK)

IAM permissions for deploying components

To deploy a component with AWS IoT Greengrass V2 you need the following permissions:

- `greengrass:ListComponents`
- `greengrass:ListComponentVersions`
- `greengrass:ListCoreDevices`
- `greengrass:CreateDeployment`
- `greengrass:GetDeployment`
- `greengrass:ListDeployments`

`CreateDeployment` and `GetDeployment` have dependent actions. For more information, see Actions defined by AWS IoT Greengrass V2.

For information about changing IAM permissions, see Changing permissions for a user.

Deploying your components (console)

Use the following procedure to deploy the client application component to a core device. The client application depends on the model component (which in turn depends on the Lookout for Vision Edge Agent). Deploying the client application component also starts the deployment of the model component and the Lookout for Vision Edge Agent.

> **Note**
>
> You can add your components to an existing deployment. You can also deploy components to a thing group.

To run this procedure, you must have a configured AWS IoT Greengrass V2 core device. For more information, see Setting up your AWS IoT Greengrass Version 2 core device.

To deploy your components to a device

1. Open the AWS IoT Greengrass console at https://console.aws.amazon.com/iot/.
2. In the left navigation pane, under **Greengrass** choose **Deployments**.
3. Under **Deployments** choose **Create**.
4. On the **Specify target** page, do the following:
   1. Under **Deployment information**, enter or modify the friendly name for your deployment.
   2. Under **Deployment target**, select **Core device** and enter a target name.
   3. Choose **Next**.
5. On the **Select components** page, do the following:
   1. Under **My components**, choose the name of your client application component (`com.lookoutvison.EdgeAgentPythonExample`).
   2. Choose **Next**
6. On the **Configure components** page, keep the current configuration and choose **Next**.
7. On the **Configure advanced settings** page, keep the current settings and choose **Next**.

8. On the **Review** page, choose **Deploy** to start deploying your component.

Checking deployment status (Console)

You can check the status of your deployment from the AWS IoT Greengrass V2 console. If your client application component uses the example recipe and code from the section called "Creating the client application component", view the client application component log after the deployment completes. If successful, the log includes a list of the Lookout for Vision models that are deployed to the component.

For information about using the AWS SDK to check deployment status, see Check deployment status.

To check deployment status

1. Open the AWS IoT Greengrass console at https://console.aws.amazon.com/iot/
2. On the left navigation pane, choose **Core devices**.
3. Under **Greengrass core devices** choose your core device.
4. Choose the **Deployments tab** to view the current deployment status.
5. After the deployments succeeds (status is **Completed**), open a terminal window on the core device and view the client application component log at `/greengrass/v2/logs/com.lookoutvison.EdgeAgentPythonExample.log`. If your deployment uses the example recipe and code, the log includes the output from `lookoutvision_edge_agent_example.py`. For example:

```
Model Details model_component:"ModelComponent"
```

Deploying the components (SDK)

Use the following procedure to deploy the client application component, model component, and the Amazon Lookout for Vision Edge Agent to your core device.

1. Create a `deployment.json` file to define the deployment configuration for your components. This file should look like the following example.

```
{
  "targetArn":"targetArn",
```

```
  "components": {
    "com.lookoutvison.EdgeAgentPythonExample": {
      "componentVersion": "1.0.0",
      "configurationUpdate": {
      }
    }
  }
}
```

- In the `targetArn` field, replace *`targetArn`* with the Amazon Resource Name (ARN) of the thing or thing group to target for the deployment, in the following format:
  - Thing: `arn:aws:iot:`*`region`*`:`*`account-id`*`:thing/`*`thingName`*
  - Thing group: `arn:aws:iot:`*`region`*`:`*`account-id:`*`thinggroup/`*`thingGroupName`*

2. Check if the deployment target has an existing deployment that you want to revise. Do the following:

   a. Run the following command to list the deployments for the deployment target. Replace `targetArn` with the Amazon Resource Name (ARN) of the target AWS IoT thing or thing group. To get the ARNs of the things in the current AWS Region, use the command `aws iot list-things`.

   ```
   aws greengrassv2 list-deployments --target-arn targetArn
   ```

   The response contains a list with the latest deployment for the target. If the response is empty, the target doesn't have an existing deployment, and you can skip to Step 3. Otherwise, copy the `deploymentId` from the response to use in the next step.

   b. Run the following command to get the deployment's details. These details include metadata, components, and job configuration. Replace `deploymentId` with the ID from the previous step.

   ```
   aws greengrassv2 get-deployment --deployment-id deploymentId
   ```

   c. Copy any of the following key-value pairs from the previous command's response into deployment.json. You can change these values for the new deployment.

   - `deploymentName` — The deployment's name.
   - `components` — The deployment's components. To uninstall a component, remove it from this object.

- `deploymentPolicies` — The deployment's policies.
- `tags` — The deployment's tags.

3. Run the following command to deploy the components on the device. Note the value of the `deploymentId` in the response.

```
aws greengrassv2 create-deployment \
    --cli-input-json file://path/to/deployment.json
```

4. Run the following command to get the status of the deployment. Change `deployment-id` to the value you noted in the previous step. The deployment has completed successfuly if the value of `deploymentStatus` is `COMPLETED`.

```
aws greengrassv2 get-deployment --deployment-id deployment-id
```

5. After the deployments succeeds, open a terminal window on the core device and view the client application component log at `/greengrass/v2/logs/com.lookoutvison.EdgeAgentPythonExample.log`. If your deployment uses the example recipe and code, the log includes the output from `lookoutvision_edge_agent_example.py`. For example:

```
Model Details model_component:"ModelComponent"
```

Amazon Lookout for Vision Edge Agent API reference

This section is the API reference for the Amazon Lookout for Vision Edge Agent.

Detecting anomalies with a model

You use the DetectAnomalies API to detect anomalies in images by using a running model on an AWS IoT Greengrass Version 2 core device.

Getting model information

APIs that get information about models deployed to a core device.

- ListModels
- DescribeModel

Running a model

APIs for starting and stopping an Amazon Lookout for Vision model that's deployed to a core device.

- StartModel
- StopModel

DetectAnomalies

Detects anomalies in the supplied image.

The response from `DetectAnomalies` includes a Boolean prediction that the image contains one or more anomalies and a confidence value for the prediction. If the model is a segmentation model, the response includes the following:

- A mask image that covers each anomaly type in a unique color. You can have `DetectAnomalies` store the mask image in shared memory, or return the mask as image bytes.
- The percentage area of the image that an anomaly type covers.
- The hex color for an anomaly type on the mask image.

> **Note**
>
> The model that you use with `DetectAnomalies` must be running. You can get the current status by calling DescribeModel. To start running a model, see StartModel.

`DetectAnomalies` supports packed bitmaps (images) in interleaved RGB888 format. The first byte represents the red channel, the second byte represents the green channel, and the third byte represents the blue channel. If you provide the image in a different format, such as BGR, the predictions from DetectAnomalies are incorrect.

By default, OpenCV uses the BGR format for image bitmaps. If you are using OpenCV to capture images for analysis with `DetectAnomalies`, you must convert the image to RGB888 format before you pass the image to `DetectAnomalies`.

The minimum supported image dimension is 64x64 pixels. The maximum supported image dimension is 4096x4096 pixels.

You can send the image in the protobuf message or through a shared memory segment. Serializing large images into the protobuf message can significantly increase the latency of calls to `DetectAnomalies`. For the least latency, we recommended that you use shared memory.

```
rpc DetectAnomalies(DetectAnomaliesRequest) returns (DetectAnomaliesResponse);
```

DetectAnomaliesRequest

The input parameters for `DetectAnomalies`.

```
message Bitmap {
  int32 width = 1;
  int32 height = 2;
  oneof data {
    bytes byte_data = 3;
    SharedMemoryHandle shared_memory_handle = 4;
  }
}
```

```
message SharedMemoryHandle {
  string name = 1;
  uint64 size = 2;
  uint64 offset = 3;
}
```

```
message AnomalyMaskParams {
SharedMemoryHandle shared_memory_handle = 2;
}
```

```
message DetectAnomaliesRequest {
string model_component = 1;
Bitmap bitmap = 2;
AnomalyMaskParams anomaly_mask_params = 3;
}
```

Bitmap

The image that you want to analyze with `DetectAnomalies`.

width

The width of the image in pixels.

height

The height of the image in pixels.

byte_data

Image bytes passed in protobuf message.

shared_memory_handle

Image bytes passed in shared memory segment.

SharedMemoryHandle

Represents a POSIX shared memory segment.

name

The name of the POSIX memory segment. For information about creating shared memory, see shm_open.

size

The image buffer size in bytes starting from the offset.

offset

The offset, in bytes, to the beginning of the image buffer from the start of the shared memory segment.

AnomalyMaskParams

Parameters for outputting an anomaly mask. (Segmentation model).

shared_memory_handle

Contains the image bytes for the mask, if `shared_memory_handle` wasn't provided.

DetectAnomaliesRequest

model_component

The name of the AWS IoT Greengrass V2 component that contains the model you want to use.

bitmap

The image that you want analyze with `DetectAnomalies`.

anomaly_mask_params

Optional parameters for outputting the mask. (Segmentation model).

DetectAnomaliesResponse

The response from `DetectAnomalies`.

```
message DetectAnomalyResult {
bool is_anomalous = 1;
float confidence = 2;
Bitmap anomaly_mask = 3;
repeated Anomaly anomalies = 4;
float anomaly_score = 5;
float anomaly_threshold = 6;
}
```

```
message Anomaly {
string name = 1;
PixelAnomaly pixel_anomaly = 2;
```

```
message PixelAnomaly {
float total_percentage_area = 1;
string hex_color = 2;
}
```

```
message DetectAnomaliesResponse {
```

```
  DetectAnomalyResult detect_anomaly_result = 1;
}
```

Anomaly

Represents an anomaly found on an image. (Segmentation model).

name

The name of an anomaly type found in an image. `name` maps to an anomaly type in the training dataset. The service automatically inserts the background anomaly type into the response from DetectAnomalies.

pixel_anomaly

Information about the pixel mask that covers an anomaly type.

PixelAnomaly

Information about the pixel mask that covers an anomaly type. (Segmentation model).

total_percentage_area

The percentage area of the image that the anomaly type covers.

hex_color

A hex color value that represents the anomaly type on the image. The color maps to the color of the anomaly type used in the training dataset.

DetectAnomalyResult

is_anomalous

Indicates if the image contains an anomaly. `true` if the image contains an anomaly. `false` if the image is normal.

confidence

The confidence that `DetectAnomalies` has in the accuracy of the prediction. `confidence` is a floating point value between 0 and 1.

anomaly_mask

if shared_memory_handle wasn't provided, contains the image bytes for the mask. (Segmentation model).

anomalies

A list of 0 or more anomalies found within the input image. (Segmentation model).

anomaly_score

A number that quantifies how much anomalies predicted for an image deviate from an image without anomalies. `anomaly_score` is a float value ranging from `0.0` to (lowest deviation from a normal image) to 1.0 (highest deviation from a normal image). Amazon Lookout for Vision returns a value for `anomaly_score`, even if the prediction for an image is normal.

anomaly_threshold

A number (float) that determines when the predicted classification for an image is normal or anomalous. Images with an `anomaly_score` that is equal to or above the value of `anomaly_threshold` are deemed anomalous. A `anomaly_score` value that is below `anomaly_threshold` indicates a normal image. The value of `anomaly_threshold` that a model uses is calculated by by Amazon Lookout for Vision when you train the model. You can't set or change the value of `anomaly_threshold`

Status codes

Code	Number	Description
OK	0	`DetectAnomalies` successfully made a predictio n
UNKNOWN	2	An unknown error has occurred.
INVALID_ARGUMENT	3	One or more input parameter s are invalid. Check the error message for more details.

Code	Number	Description
NOT_FOUND	5	A model with the specified name wasn't found.
RESOURCE_EXHAUSTED	8	There aren't enough resources to perform this operation. For example, The Lookout for Vision Edge Agent can't keep up with the rate of calls to `DetectAnomalies` . Check the error message for more details.
FAILED_PRECONDITION	9	`DetectAnomalies` was called for model that is not in the RUNNING state.
INTERNAL	13	An internal error has occurred.

DescribeModel

Describes an Amazon Lookout for Vision model that's deployed to an AWS IoT Greengrass Version 2 core device.

```
rpc DescribeModel(DescribeModelRequest) returns (DescribeModelResponse);
```

DescribeModelRequest

```
message DescribeModelRequest {
  string model_component = 1;
}
```

model_component

The name of the AWS IoT Greengrass V2 component that contains the model you want to describe.

DescribeModelResponse

```
message ModelDescription {
  string model_component = 1;
  string lookout_vision_model_arn = 2;
  ModelStatus status = 3;
  string status_message = 4;
}
```

```
message DescribeModelResponse {
  ModelDescription model_description = 1;
}
```

ModelDescription

model_component

The name of AWS IoT Greengrass Version 2 component that contains the Amazon Lookout for Vision model.

lookout_vision_model_arn

The Amazon Resource Name ARN of the Amazon Lookout for Vision model that was used to generate the AWS IoT Greengrass V2 component.

status

The current status of the model. For more information, see ModelStatus.

status_message

The status message for the model.

Status codes

Code	Number	Description
OK	0	The call was successful.

Code	Number	Description
UNKNOWN	2	An unknown error has occurred.
INVALID_ARGUMENT	3	One or more input parameter s are invalid. Check the error message for more details.
NOT_FOUND	5	A model with the supplied name wasn't found.
INTERNAL	13	An internal error has occurred.

ListModels

Lists the models deployed to an AWS IoT Greengrass Version 2 core device.

```
rpc ListModels(ListModelsRequest) returns (ListModelsResponse);
```

ListModelsRequest

```
message ListModelsRequest {}
```

ListModelsResponse

```
message ModelMetadata {
  string model_component = 1;
  string lookout_vision_model_arn = 2;
  ModelStatus status = 3;
  string status_message = 4;
}
```

```
message ListModelsResponse {
  repeated ModelMetadata models = 1;
```

```
}
```

ModelMetadata

model_component

The name of AWS IoT Greengrass Version 2 component that contains an Amazon Lookout for Vision model.

lookout_vision_model_arn

The Amazon Resource Name (ARN) of the Amazon Lookout for Vision model that was used to generate the AWS IoT Greengrass V2 component.

status

The current status of the model. For more information, see ModelStatus.

status_message

The status message for the model.

Status codes

Code	Number	Description
OK	0	The call was successful.
UNKNOWN	2	An unknown error has occurred.
INTERNAL	13	An internal error has occurred.

StartModel

Starts a model running on an AWS IoT Greengrass Version 2 core device. It might take a while for the model to start running. To check the current status call DescribeModel. The model is running if the `Status` field is `RUNNING`.

The number of models that you can run concurrently depends on the hardware specification of your core device.

```
rpc StartModel(StartModelRequest) returns (StartModelResponse);
```

StartModelRequest

```
message StartModelRequest {
  string model_component = 1;
}
```

model_component

The name of the AWS IoT Greengrass Version 2 component that contains the model you want to start.

StartModelResponse

```
message StartModelResponse {
  ModelStatus status = 1;
}
```

status

The current status of the model. The response is `STARTING` if the call succeeds. For more information, see ModelStatus.

Status codes

Code	Number	Description
OK	0	The model is starting
UNKNOWN	2	An unknown error has occurred.
INVALID_ARGUMENT	3	One or more input parameter s are invalid. Check the error message for more details.

Code	Number	Description
NOT_FOUND	5	A model with the supplied name wasn't found.
RESOURCE_EXHAUSTED	8	There isn't enough resources to perform this operation. For example, there is isn't enough memory to load the model. Check the error message for more details.
FAILED_PRECONDITION	9	The method was called for model that is not in the STOPPED or FAILED state.
INTERNAL	13	An internal error has occurred.

StopModel

Stops a model running on an AWS IoT Greengrass Version 2 core device. `StopModel` returns after the model has stopped. The model has stopped successfully if the `Status` field in the response is `STOPPED`.

```
  rpc StopModel(StopModelRequest) returns (StopModelResponse);
```

StopModelRequest

```
message StopModelRequest {
  string model_component = 1;
}
```

model_component

The name of the AWS IoT Greengrass Version 2 component that contains the model you want to stop.

StopModelResponse

```
message StopModelResponse {
  ModelStatus status = 1;
}
```

status

The current status of the model. The response is STOPPED if the call succeeds. For more information, see ModelStatus.

Status codes

Code	Number	Description
OK	0	The model is stopping.
UNKNOWN	2	An unknown error has occurred.
INVALID_ARGUMENT	3	One or more input parameter s are invalid. Check the error message for more details.
NOT_FOUND	5	A model with the supplied name wasn't found.
FAILED_PRECONDITION	9	The method was called for a model that is not in the RUNNING state.
INTERNAL	13	An internal error has occurred.

ModelStatus

The status of a model that's deployed to an AWS IoT Greengrass Version 2 core device. To get the current status, call DescribeModel.

```
enum ModelStatus {
  STOPPED = 0;
  STARTING = 1;
  RUNNING = 2;
  FAILED = 3;
  STOPPING = 4;
}
```

Using the Amazon Lookout for Vision dashboard

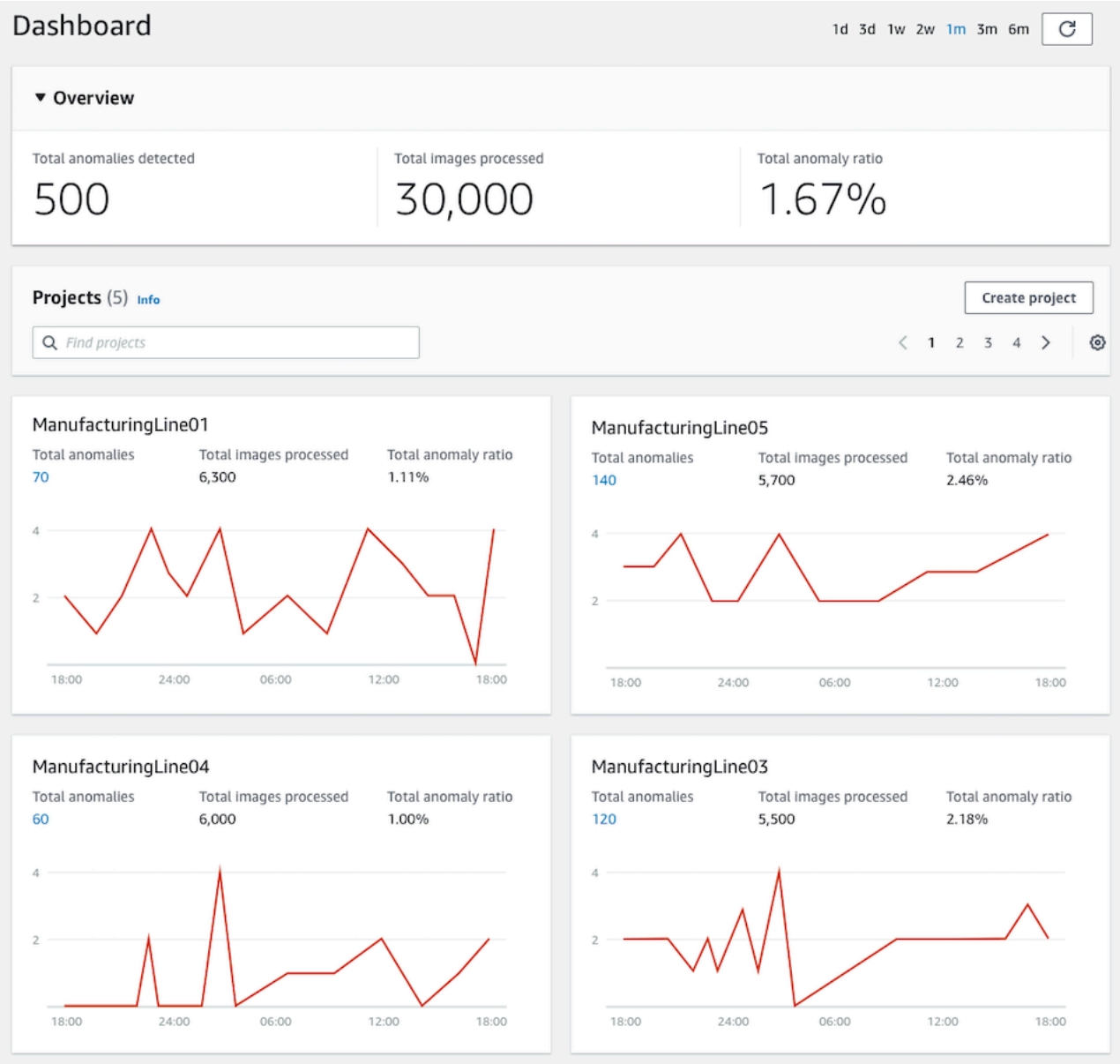
The dashboard provides an overview of metrics for your Amazon Lookout for Vision projects, such as the total number of anomalies detected over the last week. With the dashboard you get an overview for all of your projects and an overview for each individual project. You can choose the timeline over which metrics are shown. You can also use the dashboard to create a new project.

The **Overview** section shows the total number of projects, total number of images, and the total number of images detected by all of your projects.

The **Projects** section shows the following overview information for individual projects:

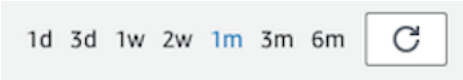
- The total number or anomalies detected.
- The total number of images processed.
- The total anomaly ratio (that is, the percentage of images detected with an anomaly).
- A graph shows the anomaly detections over the chosen time frame.

You can also get further information about a project.

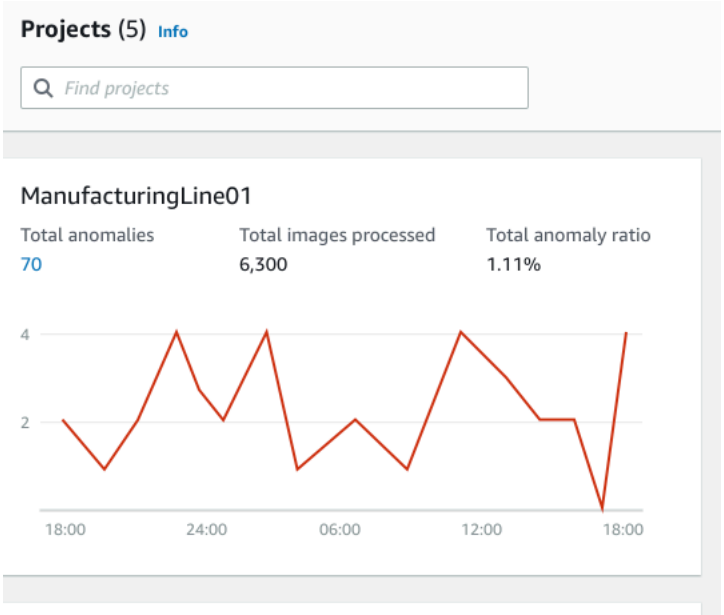


To use your dashboard

1. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
2. Choose **Get started**.
3. In the left navigation pane, choose **Dashboard**.
4. To view metrics over a specific time frame, do the following:
   a. Choose the time frame in the upper-right side of the dashboard.
   b. Choose the refresh button to show the dashboard with the new timeline.



5. To get further details about a project, choose the project name in the **Projects** section (for example, `ManufacturingLine01`).



6. To create a project, choose **Create project** in the **Projects** section.

Managing your Amazon Lookout for Vision resources

You can manage your Amazon Lookout for Vision resources by using the console or the AWS SDK. Amazon Lookout for Vision has the following resources:

- Projects
- Datasets
- Models
- Trial detections

> **Note**
>
> You can't delete a trial detection task. Also, you can't manage trial detections by using the AWS SDK.

Topics

- Viewing your projects
- Deleting a project
- Viewing your datasets
- Adding images to your dataset
- Removing images from your dataset
- Deleting a dataset
- Exporting datasets from a project (SDK)
- Viewing your models
- Deleting a model
- Tagging models
- Viewing your trial detection tasks

Viewing your projects

You can get a list of Amazon Lookout for Vision projects and information about individual projects from the console or by using the AWS SDK.

> **Note**
>
> The list of projects is eventually consistent. If you create or delete a project, you might have to wait a short while before the projects list is up to date.

Viewing your projects (console)

Perform the steps in the following procedure to view your projects in the console.

To view your projects

1. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
2. Choose **Get started**.
3. In the left navigation pane, choose **Projects**. The projects view is shown.
4. Choose a project name to see the project's details.

Viewing your projects (SDK)

A project manages the datasets and models for a single use case. For example, detecting anomalies in machine parts. The following example calls `ListProjects` to get a list of your projects.

To view your projects (SDK)

1. If you haven't already done so, install and configure the AWS CLI and the AWS SDKs. For more information, see Step 4: Set up the AWS CLI and AWS SDKs.
2. Use the following example code to view your projects.

   CLI

   Use the `list-projects` command to list the projects in your account.

   ```
   aws lookoutvision list-projects \
     --profile lookoutvision-access
   ```

   Use the `describe-project` command to get information about a project.

Change the value of `project-name` to the name of the project that you want to describe.

```
aws lookoutvision describe-project --project-name project_name \
  --profile lookoutvision-access
```

Python

This code is taken from the AWS Documentation SDK examples GitHub repository. See the full example here.

```
    @staticmethod
    def list_projects(lookoutvision_client):
        """
        Lists information about the projects that are in in your AWS account
        and in the current AWS Region.

        :param lookoutvision_client: A Boto3 Lookout for Vision client.
        """
        try:
            response = lookoutvision_client.list_projects()
            for project in response["Projects"]:
                print("Project: " + project["ProjectName"])
                print("\tARN: " + project["ProjectArn"])
                print("\tCreated: " + str(["CreationTimestamp"]))
                print("Datasets")
                project_description = lookoutvision_client.describe_project(
                    ProjectName=project["ProjectName"]
                )
                if not project_description["ProjectDescription"]["Datasets"]:
                    print("\tNo datasets")
                else:
                    for dataset in project_description["ProjectDescription"][
                        "Datasets"
                    ]:
                        print(f"\ttype: {dataset['DatasetType']}")
                        print(f"\tStatus: {dataset['StatusMessage']}")

                print("Models")
                response_models = lookoutvision_client.list_models(
                    ProjectName=project["ProjectName"]
                )
                if not response_models["Models"]:
```

```
                    print("\tNo models")
                else:
                    for model in response_models["Models"]:
                        Models.describe_model(
                            lookoutvision_client,
                            project["ProjectName"],
                            model["ModelVersion"],
                        )


 print("------------------------------------------------------------\n")
            print("Done!")
        except ClientError:
            logger.exception("Problem listing projects.")
            raise
```

Java V2

This code is taken from the AWS Documentation SDK examples GitHub repository. See the full example here.

```
/**
 * Lists the Amazon Lookout for Vision projects in the current AWS account and
 AWS
 * Region.
 *
 * @param lfvClient   An Amazon Lookout for Vision client.
 * @param projectName The name of the project that you want to create.
 * @return List<ProjectMetadata> Metadata for each project.
 */
public static List<ProjectMetadata> listProjects(LookoutVisionClient lfvClient)
                throws LookoutVisionException {

        logger.log(Level.INFO, "Getting projects:");
        ListProjectsRequest listProjectsRequest = ListProjectsRequest.builder()
                        .maxResults(100)
                        .build();

        List<ProjectMetadata> projectMetadata = new ArrayList<>();
```

```
        ListProjectsIterable projects =
 lfvClient.listProjectsPaginator(listProjectsRequest);

        projects.stream().flatMap(r -> r.projects().stream())
                        .forEach(project -> {
                                projectMetadata.add(project);
                                logger.log(Level.INFO, project.projectName());
                        });

        logger.log(Level.INFO, "Finished getting projects.");

        return projectMetadata;

}
```

Deleting a project

You can delete a project from the projects view page in the console or by using the `DeleteProject` operation.

The images referenced by a project's datasets aren't deleted.

Deleting a project (console)

Use the following procedure to delete a project. If you use the console procedure, associated model versions and datasets are deleted for you.

To delete a project

1. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
2. Choose **Get started**.
3. In the left navigation pane, choose **Projects**.
4. On the **Projects** page, select the project that you want to delete.
5. Choose **Delete** at the top of the page.
6. In the **Delete** dialog box, enter **delete** to confirm that you want to delete the project.
7. If necessary, choose to delete any associated datasets and models.
8. Choose **Delete project**.

Deleting a project (SDK)

You delete an Amazon Lookout for Vision project by calling DeleteProject and supplying the name of the project that you want to delete.

Before you can delete a project, you must first delete all models in the project. For more information, see Deleting a model (SDK). You also have to delete the datasets associated with the model. For more information, see Deleting a dataset.

The project might take a few moments to delete. During that time, the status of the project is `DELETING`. The project is deleted if a subsequent call to `DeleteProject` doesn't include the project that you deleted.

To delete a project (SDK)

1. If you haven't already done so, install and configure the AWS CLI and the AWS SDKs. For more information, see Step 4: Set up the AWS CLI and AWS SDKs.
2. Use the following code to delete a project.

   AWS CLI

   Change the value of `project-name` to the name of the project that you want to delete.

   ```
   aws lookoutvision delete-project --project-name project_name \
     --profile lookoutvision-access
   ```

   Python

   This code is taken from the AWS Documentation SDK examples GitHub repository. See the full example here.

   ```
       @staticmethod
       def delete_project(lookoutvision_client, project_name):
           """
           Deletes a Lookout for Vision Model

           :param lookoutvision_client: A Boto3 Lookout for Vision client.
           :param project_name: The name of the project that you want to delete.
           """
           try:
   ```

```
            logger.info("Deleting project: %s", project_name)
            response =
 lookoutvision_client.delete_project(ProjectName=project_name)
            logger.info("Deleted project ARN: %s ", response["ProjectArn"])
        except ClientError as err:
            logger.exception("Couldn't delete project %s.", project_name)
            raise
```

Java V2

This code is taken from the AWS Documentation SDK examples GitHub repository. See the full example here.

```
/**
 * Deletes an Amazon Lookout for Vision project.
 *
 * @param lfvClient   An Amazon Lookout for Vision client.
 * @param projectName The name of the project that you want to create.
 * @return String The ARN of the deleted project.
 */
public static String deleteProject(LookoutVisionClient lfvClient, String
 projectName)
                throws LookoutVisionException {

        logger.log(Level.INFO, "Deleting project: {0}", projectName);

        DeleteProjectRequest deleteProjectRequest =
 DeleteProjectRequest.builder()
                        .projectName(projectName)
                        .build();

        DeleteProjectResponse response =
 lfvClient.deleteProject(deleteProjectRequest);

        logger.log(Level.INFO, "Deleted project: {0} ARN: {1}",
                        new Object[] { projectName, response.projectArn() });

        return response.projectArn();

}
```

Viewing your datasets

A project can have a single dataset that's used for training and testing your model. Alternatively, You can have separate training and test datasets. You can use the console to view your datasets. You can also use the `DescribeDataset` operation to get information about a dataset (training or test).

Viewing the datasets in a project (console)

Perform the steps in the following procedure to view your project's datasets in the console.

To view your datasets (console)

1. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
2. Choose **Get started**.
3. In the left navigation pane, choose **Projects**.
4. On the **Projects** page, select the project that contains the datasets that you want to view.
5. In the left navigation pane, choose **Dataset** to view the dataset details. If you have a training and a test dataset, a tab for each dataset is shown.

Viewing the datasets in a project (SDK)

You can use the `DescribeDataset` operation to get information about the training or test dataset associated with a project.

To view your datasets (SDK)

1. If you haven't already done so, install and configure the AWS CLI and the AWS SDKs. For more information, see Step 4: Set up the AWS CLI and AWS SDKs.
2. Use the following example code to view a dataset.

   CLI

   Change the following values:

   - `project-name` to the name of the project that contains the model that you want to view.

- `dataset-type` to the type of dataset that you want to view (`train` or `test`).

```
aws lookoutvision describe-dataset --project-name project name\
  --dataset-type train or test \
  --profile lookoutvision-access
```

Python

This code is taken from the AWS Documentation SDK examples GitHub repository. See the full example here.

```
    @staticmethod
    def describe_dataset(lookoutvision_client, project_name, dataset_type):
        """
        Gets information about a Lookout for Vision dataset.

        :param lookoutvision_client: A Boto3 Lookout for Vision client.
        :param project_name: The name of the project that contains the dataset
 that
                             you want to describe.
        :param dataset_type: The type (train or test) of the dataset that you
 want
                             to describe.
        """
        try:
            response = lookoutvision_client.describe_dataset(
                ProjectName=project_name, DatasetType=dataset_type
            )
            print(f"Name: {response['DatasetDescription']['ProjectName']}")
            print(f"Type: {response['DatasetDescription']['DatasetType']}")
            print(f"Status: {response['DatasetDescription']['Status']}")
            print(f"Message: {response['DatasetDescription']['StatusMessage']}")
            print(f"Images: {response['DatasetDescription']['ImageStats']
['Total']}")
            print(f"Labeled: {response['DatasetDescription']['ImageStats']
['Labeled']}")
            print(f"Normal: {response['DatasetDescription']['ImageStats']
['Normal']}")
            print(f"Anomaly: {response['DatasetDescription']['ImageStats']
['Anomaly']}")
        except ClientError:
```

```
            logger.exception("Service error: problem listing datasets.")
            raise
        print("Done.")
```

Java V2

This code is taken from the AWS Documentation SDK examples GitHub repository. See the full example here.

```
/**
 * Gets the description for a Amazon Lookout for Vision dataset.
 *
 * @param lfvClient   An Amazon Lookout for Vision client.
 * @param projectName The name of the project in which you want to describe a
 *                    dataset.
 * @param datasetType The type of the dataset that you want to describe (train
 *                    or test).
 * @return DatasetDescription A description of the dataset.
 */
public static DatasetDescription describeDataset(LookoutVisionClient lfvClient,
                String projectName,
                String datasetType) throws LookoutVisionException {

        logger.log(Level.INFO, "Describing {0} dataset for project {1}",
                        new Object[] { datasetType, projectName });

        DescribeDatasetRequest describeDatasetRequest =
 DescribeDatasetRequest.builder()
                        .projectName(projectName)
                        .datasetType(datasetType)
                        .build();

        DescribeDatasetResponse describeDatasetResponse =
 lfvClient.describeDataset(describeDatasetRequest);
        DatasetDescription datasetDescription =
 describeDatasetResponse.datasetDescription();

        logger.log(Level.INFO, "Project: {0}\n"
                        + "Created: {1}\n"
                        + "Type: {2}\n"
                        + "Total: {3}\n"
                        + "Labeled: {4}\n"
```

```
                        + "Normal: {5}\n"
                        + "Anomalous: {6}\n",
                        new Object[] {
                                        datasetDescription.projectName(),
                                        datasetDescription.creationTimestamp(),
                                        datasetDescription.datasetType(),

 datasetDescription.imageStats().total().toString(),

 datasetDescription.imageStats().labeled().toString(),

 datasetDescription.imageStats().normal().toString(),

 datasetDescription.imageStats().anomaly().toString(),
                        });

        return datasetDescription;

}
```

Adding images to your dataset

After you create a dataset, you might want to add more images to the dataset. For example, if model evaluation indicates a poor model, you can enhance the quality of your model by adding more images. If you have created a test dataset, adding more images can increase the accuracy of your model's performance metrics.

Retrain your model after updating your datasets.

Topics

- Adding more images
- Adding more images (SDK)

Adding more images

You can add more images to your datasets by uploading images from your local computer. To add more labeled images with the SDK, use the UpdateDatasetEntries operation.

To add more images to your dataset (console)

1. Choose **Actions** and select the dataset that you want to add images to.
2. Choose the images you want to upload to the dataset. You can drag the images or choose the images that you want to upload from your local computer. You can upload up to 30 images at a time.
3. Choose **Upload images**.
4. Choose **Save changes**.

When you are done adding more images, you need to label them so that they can be used to train the model. For more information, see Classifying images (console).

Adding more images (SDK)

To add more labeled images with the SDK, use the UpdateDatasetEntries operation. You supply a manifest file that contains the images that you want to add. You can also update existing images by specifying the image in the `source-ref` field of the JSON line in the manifest file. For more information, see Creating a manifest file.

To add more images to a dataset (SDK)

1. If you haven't already done so, install and configure the AWS CLI and the AWS SDKs. For more information, see Step 4: Set up the AWS CLI and AWS SDKs.
2. Use the following example code to add more images to a dataset.

   CLI

   Change the following values:

   - `project-name` to the name of the project that contains the dataset you want to update.
   - `dataset-type` to the type of dataset that you want to update (`train` or `test`).
   - `changes` to the location the manifest file that contain dataset updates.

```
aws lookoutvision update-dataset-entries\
    --project-name project\
    --dataset-type train or test\
    --changes fileb://manifest file \
```

```
    --profile lookoutvision-access
```

Python

This code is taken from the AWS Documentation SDK examples GitHub repository. See the full example here.

```
    @staticmethod
    def update_dataset_entries(lookoutvision_client, project_name, dataset_type,
 updates_file):
        """
        Adds dataset entries to an Amazon Lookout for Vision dataset.
        :param lookoutvision_client: The Amazon Rekognition Custom Labels Boto3
 client.
        :param project_name: The project that contains the dataset that you want
 to update.
        :param dataset_type: The type of the dataset that you want to update
 (train or test).
        :param updates_file: The manifest file of JSON Lines that contains the
 updates.
        """

        try:
            status = ""
            status_message = ""
            manifest_file = ""

            # Update dataset entries
            logger.info(f"""Updating {dataset_type} dataset for project
 {project_name}
with entries from {updates_file}.""")

            with open(updates_file) as f:
                manifest_file = f.read()

            lookoutvision_client.update_dataset_entries(
                ProjectName=project_name,
                DatasetType=dataset_type,
                Changes=manifest_file,
            )

            finished = False
            while finished == False:
```

```
                dataset =
lookoutvision_client.describe_dataset(ProjectName=project_name,

DatasetType=dataset_type)

                status = dataset['DatasetDescription']['Status']
                status_message = dataset['DatasetDescription']['StatusMessage']

                if status == "UPDATE_IN_PROGRESS":
                    logger.info(
                        (f"Updating {dataset_type} dataset for project
{project_name}."))
                    time.sleep(5)
                    continue

                if status == "UPDATE_FAILED_ROLLBACK_IN_PROGRESS":
                    logger.info(
                        (f"Update failed, rolling back {dataset_type} dataset
for project {project_name}."))
                    time.sleep(5)
                    continue

                if status == "UPDATE_COMPLETE":
                    logger.info(
                        f"Dataset updated: {status} : {status_message} :
{dataset_type} dataset for project {project_name}.")
                    finished = True
                    continue

                if status == "UPDATE_FAILED_ROLLBACK_COMPLETE":
                    logger.info(
                        f"Rollback complated after update failure: {status} :
{status_message} : {dataset_type} dataset for project {project_name}.")
                    finished = True
                    continue

                logger.exception(
                    f"Failed. Unexpected state for dataset update: {status} :
{status_message} : {dataset_type} dataset for project {project_name}.")
                raise Exception(
                    f"Failed. Unexpected state for dataset update: {status} :
{status_message} :{dataset_type} dataset for project {project_name}.")
```

```
            logger.info(f"Added entries to dataset.")

            return status, status_message

        except ClientError as err:
            logger.exception(
                f"Couldn't update dataset: {err.response['Error']['Message']}")
            raise
```

Java V2

This code is taken from the AWS Documentation SDK examples GitHub repository. See the full example here.

```
/**
 * Updates an Amazon Lookout for Vision dataset from a manifest file.
 * Returns after Lookout for Vision updates the dataset.
 *
 * @param lfvClient    An Amazon Lookout for Vision client.
 * @param projectName  The name of the project in which you want to update a
 *                     dataset.
 * @param datasetType  The type of the dataset that you want to update (train or
 *                     test).
 * @param manifestFile The name and location of a local manifest file that you
 want to
 * use to update the dataset.
 * @return DatasetStatus The status of the updated dataset.
 */

public static DatasetStatus updateDatasetEntries(LookoutVisionClient lfvClient,
 String projectName,
                String datasetType, String updateFile) throws
 FileNotFoundException, LookoutVisionException,
                InterruptedException {

        logger.log(Level.INFO, "Updating {0} dataset for project {1}",
                        new Object[] { datasetType, projectName });

        InputStream sourceStream = new FileInputStream(updateFile);
        SdkBytes sourceBytes = SdkBytes.fromInputStream(sourceStream);

        UpdateDatasetEntriesRequest updateDatasetEntriesRequest =
 UpdateDatasetEntriesRequest.builder()
```

```
                        .projectName(projectName)
                        .datasetType(datasetType)
                        .changes(sourceBytes)
                        .build();

        lfvClient.updateDatasetEntries(updateDatasetEntriesRequest);

        boolean finished = false;
        DatasetStatus status = null;

        // Wait until update completes.

        do {

                DescribeDatasetRequest describeDatasetRequest =
DescribeDatasetRequest.builder()
                                .projectName(projectName)
                                .datasetType(datasetType)
                                .build();
                DescribeDatasetResponse describeDatasetResponse = lfvClient
                                .describeDataset(describeDatasetRequest);

                DatasetDescription datasetDescription =
describeDatasetResponse.datasetDescription();

                status = datasetDescription.status();

                switch (status) {

                        case UPDATE_COMPLETE:
                                logger.log(Level.INFO, "{0} Dataset updated for
project {1}.",
                                                new Object[] { datasetType,
projectName });
                                finished = true;
                                break;

                        case UPDATE_IN_PROGRESS:

                                logger.log(Level.INFO, "{0} Dataset update for
project {1} in progress.",
                                                new Object[] { datasetType,
projectName });
                                TimeUnit.SECONDS.sleep(5);
```

```
                                break;

                        case UPDATE_FAILED_ROLLBACK_IN_PROGRESS:
                                logger.log(Level.SEVERE,
                                                "{0} Dataset update failed for
 project {1}. Rolling back",
                                                new Object[] { datasetType,
 projectName });

                                TimeUnit.SECONDS.sleep(5);

                                break;

                        case UPDATE_FAILED_ROLLBACK_COMPLETE:
                                logger.log(Level.SEVERE,
                                                "{0} Dataset update failed for
 project {1}. Rollback completed.",
                                                new Object[] { datasetType,
 projectName });
                                finished = true;
                                break;

                        default:
                                logger.log(Level.SEVERE,
                                                "{0} Dataset update failed for
 project {1}. Unexpected error returned.",
                                                new Object[] { datasetType,
 projectName });
                                finished = true;

                }

        } while (!finished);

        return status;

}
```

3. Repeat the previous step and provide values for the other dataset type.

Removing images from your dataset

You can't delete images directly from a dataset. Instead you must delete the existing dataset and create a new dataset without the images that you want to remove. How you remove images depends how you imported images into the existing dataset (manifest file, Amazon S3 bucket, or local computer).

You can also use the AWS SDK to remove images. This is useful when creating an image segmentation model without an image segmentation manifest file, making it unnecessary to redraw the image masks with the Amazon Lookout for Vision console.

Topics

- Removing images from a dataset (Console)
- Removing images from a dataset (SDK)

Removing images from a dataset (Console)

Use the following procedure to remove images from a dataset with the Amazon Lookout for Vision console.

To remove images from a dataset (console)

1. Open the project's dataset gallery.
2. Note the name of each image that you want to remove.
3. Delete the existing dataset.
4. Do one of the following:
   - If you created the dataset with a manifest file, do:
     a. In a text editor, open the manifest file that you used to create the dataset.
     b. Remove the JSON line for each image that you noted in step 2. You can identify the JSON line for an image by checking the `source-ref` field.
     c. Save the manifest file.
     d. Create a new dataset with the updated manifest file.
   - If you created the dataset from images imported from an Amazon S3 bucket, do:
     a. Delete the images you noted in step 2 from the Amazon S3 bucket.

   b. Create a new dataset with the remaining images in the Amazon S3 bucket. If you classify images by folder name, you don't need to classify images in the next step.
   c. Do one of the following:
      - If you are creating an image classification model, classify each unlabeled image.
      - If you are creating an image segmentation model, classify and segment each unlabeled image.
- If you created the dataset from images imported from a local computer, do:
   a. On your computer, create a folder with the images that you want to use. Don't include the images you want to remove from the dataset. For more information, see Creating a dataset using images stored on your local computer.
   b. Create the dataset with the images in the folder that you created in step 4.a.
   c. Do one of the following:
      - If you are creating an image classification model, classify each unlabeled image.
      - If you are creating an image segmentation model, classify and segment each unlabeled image.

5. Train the model.

Removing images from a dataset (SDK)

You can use the AWS SDK to remove images from a dataset.

To remove images from a dataset (SDK)

1. Open the project's dataset gallery.
2. Note the name of each image that you want to remove.
3. Export the JSON lines for the dataset by using the ListDatasetEntries operation.
4. Create a manifest file with the exported JSON lines.
5. In a text editor, open the manifest file.
6. Remove the JSON line for each image that you noted in step 2. You can identify the JSON line for an image by checking the `source-ref` field.
7. Save the manifest file.
8. Delete the existing dataset.
9. Create a new dataset with the updated manifest file.

10. Train the model.

Deleting a dataset

You can delete a dataset from a project by using the console or the `DeleteDataset` operation. The images referenced by a dataset aren't deleted. If you delete the test dataset from a project that has a training and a test dataset, the project reverts to a single dataset project—the remaining dataset is split during training to create a training and test dataset. If you delete the training dataset, you can't train a model in the project until you create a new training dataset.

Deleting a dataset (console)

Perform the steps in the following procedure to delete a dataset. If you delete all of the datasets in a project, the **Create dataset** page is displayed.

To delete a dataset (console)

1. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
2. Choose **Get started**.
3. In the left navigation pane, choose **Projects**.
4. On the **Projects** page, select the project that contains the dataset that you want to delete.
5. In the left navigation pane, choose **Dataset**.
6. Choose **Actions** and then select the dataset that you want to delete.
7. In the **Delete** dialog box, enter **delete** to confirm that you want to delete the dataset.
8. Choose **Delete training dataset** or **Delete test dataset** to delete the dataset.

Deleting a dataset (SDK)

Use the `DeleteDataset` operation to delete a dataset.

To delete a dataset (SDK)

1. If you haven't already done so, install and configure the AWS CLI and the AWS SDKs. For more information, see Step 4: Set up the AWS CLI and AWS SDKs.
2. Use the following example code to delete a model.

CLI

Change the value of the following

- `project-name` to the name of the project that contains the model that you want to delete.
- `dataset-type` to either `train` or `test`, depending on which dataset you want to delete. If you have a single dataset project, specify `train` to delete the dataset.

```
aws lookoutvision delete-dataset --project-name project name\
  --dataset-type dataset type \
  --profile lookoutvision-access
```

Python

This code is taken from the AWS Documentation SDK examples GitHub repository. See the full example here.

```
    @staticmethod
    def delete_dataset(lookoutvision_client, project_name, dataset_type):
        """
        Deletes a Lookout for Vision dataset

        :param lookoutvision_client: A Boto3 Lookout for Vision client.
        :param project_name: The name of the project that contains the dataset
that
                             you want to delete.
        :param dataset_type: The type (train or test) of the dataset that you
                             want to delete.
        """
        try:
            logger.info(
                "Deleting the %s dataset for project %s.", dataset_type,
project_name
            )
            lookoutvision_client.delete_dataset(
                ProjectName=project_name, DatasetType=dataset_type
            )
            logger.info("Dataset deleted.")
        except ClientError:
```

```
            logger.exception("Service error: Couldn't delete dataset.")
            raise
```

Java V2

This code is taken from the AWS Documentation SDK examples GitHub repository. See the full example here.

```
/**
 * Deletes the train or test dataset in an Amazon Lookout for Vision project.
 *
 * @param lfvClient   An Amazon Lookout for Vision client.
 * @param projectName The name of the project in which you want to delete a
 *                    dataset.
 * @param datasetType The type of the dataset that you want to delete (train or
 *                    test).
 * @return Nothing.
 */
public static void deleteDataset(LookoutVisionClient lfvClient, String
 projectName, String datasetType)
                throws LookoutVisionException {

        logger.log(Level.INFO, "Deleting {0} dataset for project {1}",
                        new Object[] { datasetType, projectName });

        DeleteDatasetRequest deleteDatasetRequest =
 DeleteDatasetRequest.builder()
                        .projectName(projectName)
                        .datasetType(datasetType)
                        .build();

        lfvClient.deleteDataset(deleteDatasetRequest);

        logger.log(Level.INFO, "Deleted {0} dataset for project {1}",
                        new Object[] { datasetType, projectName });

}
```

Exporting datasets from a project (SDK)

You can use the AWS SDK to export datasets from an Amazon Lookout for Vision project to an Amazon S3 bucket location.

By exporting a dataset, you can do tasks such as creating a Lookout for Vision project with a copy of a source project's datasets. You also can create a snapshot of the datasets used for a specific version of a model.

The Python code in this procedure exports the training dataset (manifest and dataset images) for a project to a destination Amazon S3 location that you specify. If present in the project, the code also exports the test dataset manifest and dataset images. The destination can be in the same Amazon S3 bucket as the source project, or a different Amazon S3 bucket. The code uses the ListDatasetEntries operation to get the dataset manifest files. Amazon S3 operations copy the dataset images and updated manifest files to the destination Amazon S3 location.

This procedure shows how to export a project's datasets. It also shows how to create a new project with the exported datasets.

To export the datasets from a project (SDK)

1. If you haven't already done so, install and configure the AWS CLI and the AWS SDKs. For more information, see Step 4: Set up the AWS CLI and AWS SDKs.
2. Determine the destination Amazon S3 path for the dataset export. Be sure that the destination is in an AWS Region that Amazon Lookout for Vision supports. To create a new Amazon S3 bucket, see Creating a bucket.
3. Make sure the user has access permissions to the destination Amazon S3 path for the dataset export and the S3 locations for the image files in the source project datasets. You can use the following policy which assumes the images files can be in any location. Replace *bucket/path* with the destination bucket and path for the dataset export.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "PutExports",
            "Effect": "Allow",
            "Action": [
                "S3:PutObjectTagging",
```

```
                "S3:PutObject"
            ],
            "Resource": "arn:aws:s3:::bucket/path/*"
        },
        {
            "Sid": "GetSourceRefs",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:GetObjectTagging",
                "s3:GetObjectVersion"
            ],
            "Resource": "*"
        }
    ]
}
```

To provide access, add permissions to your users, groups, or roles:

- Users and groups in AWS IAM Identity Center:

  Create a permission set. Follow the instructions in Create a permission set in the *AWS IAM Identity Center User Guide*.

- Users managed in IAM through an identity provider:

  Create a role for identity federation. Follow the instructions in Create a role for a third-party identity provider (federation) in the *IAM User Guide*.

- IAM users:
  - Create a role that your user can assume. Follow the instructions in Create a role for an IAM user in the *IAM User Guide*.
  - (Not recommended) Attach a policy directly to a user or add a user to a user group. Follow the instructions in Adding permissions to a user (console) in the *IAM User Guide*.

4. Save the following code to a file named `dataset_export.py`.

```
"""
Purpose

Shows how to export the datasets (manifest files and images)
from an Amazon Lookout for Vision project to a new Amazon
```

```
S3 location.
"""

import argparse
import json
import logging

import boto3
from botocore.exceptions import ClientError

logger = logging.getLogger(__name__)


def copy_file(s3_resource, source_file, destination_file):
    """
    Copies a file from a source Amazon S3 folder to a destination
    Amazon S3 folder.
    The destination can be in a different S3 bucket.
    :param s3: An Amazon S3 Boto3 resource.
    :param source_file: The Amazon S3 path to the source file.
    :param destination_file: The destination Amazon S3 path for
    the copy operation.
    """

    source_bucket, source_key = source_file.replace("s3://", "").split("/", 1)
    destination_bucket, destination_key = destination_file.replace("s3://",
 "").split(
        "/", 1
    )

    try:
        bucket = s3_resource.Bucket(destination_bucket)
        dest_object = bucket.Object(destination_key)
        dest_object.copy_from(CopySource={"Bucket": source_bucket, "Key":
 source_key})
        dest_object.wait_until_exists()
        logger.info("Copied %s to %s", source_file, destination_file)
    except ClientError as error:
        if error.response["Error"]["Code"] == "404":
            error_message = (
                f"Failed to copy {source_file} to "
                f"{destination_file}. : {error.response['Error']['Message']}"
            )
            logger.warning(error_message)
```

```
            error.response["Error"]["Message"] = error_message
        raise


def upload_manifest_file(s3_resource, manifest_file, destination):
    """
    Uploads a manifest file to a destination Amazon S3 folder.
    :param s3: An Amazon S3 Boto3 resource.
    :param manifest_file: The manifest file that you want to upload.
    :destination: The Amazon S3 folder location to upload the manifest
    file to.
    """

    destination_bucket, destination_key = destination.replace("s3://",
 "").split("/", 1)

    bucket = s3_resource.Bucket(destination_bucket)

    put_data = open(manifest_file, "rb")
    obj = bucket.Object(destination_key + manifest_file)

    try:
        obj.put(Body=put_data)
        obj.wait_until_exists()
        logger.info("Put manifest file '%s' to bucket '%s'.", obj.key,
 obj.bucket_name)
    except ClientError:
        logger.exception(
            "Couldn't put manifest file '%s' to bucket '%s'.", obj.key,
 obj.bucket_name
        )
        raise
    finally:
        if getattr(put_data, "close", None):
            put_data.close()


def get_dataset_types(lookoutvision_client, project):
    """
    Determines the types of the datasets (train or test) in an
    Amazon Lookout for Vision project.
    :param lookoutvision_client: A Lookout for Vision Boto3 client.
    :param project: The Lookout for Vision project that you want to check.
    :return: The dataset types in the project.
```

```
    """

    try:
        response = lookoutvision_client.describe_project(ProjectName=project)

        datasets = []

        for dataset in response["ProjectDescription"]["Datasets"]:
            if dataset["Status"] in ("CREATE_COMPLETE", "UPDATE_COMPLETE"):
                datasets.append(dataset["DatasetType"])
        return datasets

    except lookoutvision_client.exceptions.ResourceNotFoundException:
        logger.exception("Project %s not found.", project)
        raise


def process_json_line(s3_resource, entry, dataset_type, destination):
    """
    Creates a JSON line for a new manifest file, copies image and mask to
    destination.
    :param s3_resource: An Amazon S3 Boto3 resource.
    :param entry: A JSON line from the manifest file.
    :param dataset_type: The type (train or test) of the dataset that
    you want to create the manifest file for.
    :param destination: The destination Amazon S3 folder for the manifest
    file and dataset images.
    :return: A JSON line with details for the destination location.
    """
    entry_json = json.loads(entry)

    print(f"source: {entry_json['source-ref']}")

    # Use existing folder paths to ensure console added image names don't clash.
    bucket, key = entry_json["source-ref"].replace("s3://", "").split("/", 1)
    logger.info("Source location: %s/%s", bucket, key)

    destination_image_location = destination + dataset_type + "/images/" + key

    copy_file(s3_resource, entry_json["source-ref"], destination_image_location)

    # Update JSON for writing.
    entry_json["source-ref"] = destination_image_location
```

```
    if "anomaly-mask-ref" in entry_json:
        source_anomaly_ref = entry_json["anomaly-mask-ref"]
        mask_bucket, mask_key = source_anomaly_ref.replace("s3://", "").split("/",
 1)

        destination_mask_location = destination + dataset_type + "/masks/" +
 mask_key
        entry_json["anomaly-mask-ref"] = destination_mask_location

        copy_file(s3_resource, source_anomaly_ref, entry_json["anomaly-mask-ref"])

    return entry_json


def write_manifest_file(
    lookoutvision_client, s3_resource, project, dataset_type, destination
):
    """
    Creates a manifest file for a dataset. Copies the manifest file and
    dataset images (and masks, if present) to the specified Amazon S3 destination.
    :param lookoutvision_client: A Lookout for Vision Boto3 client.
    :param project: The Lookout for Vision project that you want to use.
    :param dataset_type: The type (train or test) of the dataset that
    you want to create the manifest file for.
    :param destination: The destination Amazon S3 folder for the manifest file
    and dataset images.
    """

    try:
        # Create a reusable Paginator
        paginator = lookoutvision_client.get_paginator("list_dataset_entries")

        # Create a PageIterator from the Paginator
        page_iterator = paginator.paginate(
            ProjectName=project,
            DatasetType=dataset_type,
            PaginationConfig={"PageSize": 100},
        )

        output_manifest_file = dataset_type + ".manifest"

        # Create manifest file then upload to Amazon S3 with images.
        with open(output_manifest_file, "w", encoding="utf-8") as manifest_file:
            for page in page_iterator:
```

```
                for entry in page["DatasetEntries"]:
                    try:
                        entry_json = process_json_line(
                            s3_resource, entry, dataset_type, destination
                        )

                        manifest_file.write(json.dumps(entry_json) + "\n")

                    except ClientError as error:
                        if error.response["Error"]["Code"] == "404":
                            print(error.response["Error"]["Message"])
                            print(f"Excluded JSON line: {entry}")
                        else:
                            raise
        upload_manifest_file(
            s3_resource, output_manifest_file, destination + "datasets/"
        )

    except ClientError:
        logger.exception("Problem getting dataset_entries")
        raise


def export_datasets(lookoutvision_client, s3_resource, project, destination):
    """
    Exports the datasets from an Amazon Lookout for Vision project to a specified
    Amazon S3 destination.
    :param project: The Lookout for Vision project that you want to use.
    :param destination: The destination Amazon S3 folder for the exported datasets.
    """
    # Add trailing backslash, if missing.
    destination = destination if destination[-1] == "/" else destination + "/"

    print(f"Exporting project {project} datasets to {destination}.")

    # Get each dataset and export to destination.

    dataset_types = get_dataset_types(lookoutvision_client, project)
    for dataset in dataset_types:
        logger.info("Copying %s dataset to %s.", dataset, destination)

        write_manifest_file(
            lookoutvision_client, s3_resource, project, dataset, destination
        )
```

```
    print("Exported dataset locations")
    for dataset in dataset_types:
        print(f"   {dataset}: {destination}datasets/{dataset}.manifest")

    print("Done.")


def add_arguments(parser):
    """
    Adds command line arguments to the parser.
    :param parser: The command line parser.
    """

    parser.add_argument("project", help="The project that contains the dataset.")
    parser.add_argument("destination", help="The destination Amazon S3 folder.")


def main():
    """
    Exports the datasets from an Amazon Lookout for Vision project to a
    destination Amazon S3 location.
    """
    logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")
    parser = argparse.ArgumentParser(usage=argparse.SUPPRESS)
    add_arguments(parser)

    args = parser.parse_args()

    try:
        session = boto3.Session(profile_name="lookoutvision-access")
        lookoutvision_client = session.client("lookoutvision")
        s3_resource = session.resource("s3")

        export_datasets(
            lookoutvision_client, s3_resource, args.project, args.destination
        )
    except ClientError as err:
        logger.exception(err)
        print(f"Failed: {format(err)}")


if __name__ == "__main__":
    main()
```

5. Run the code. Supply the following command line arguments:

   - project – The name of the source project that contains the datasets that you want to export.
   - destination – The destination Amazon S3 path for the datasets.

   For example, `python dataset_export.py myproject s3://bucket/path/`

6. Note the manifest file locations that the code displays. You need them in step 8.
7. Create a new Lookout for Vision project with exported dataset by following the instructions at Creating your project.
8. Do one of the following:

   - Use the Lookout for Vision console to create datasets for your new project by following the instructions at Creating a dataset with a manifest file (console). You don't need to do steps 1–6.

     For step 12, do the following:

     a. If the source project has a test dataset, choose **Separate training and test datasests**, otherwise choose **single dataset**.

     b. For **.manifest file location**, enter the location of the appropriate manifest file (train or test) that you noted in step 6.

   - Use the CreateDataset operation to create datasets for your new project by using the code at Creating a dataset with a manifest file (SDK). For the `manifest_file` parameter, use the manifest file location you noted in step 6. If the source project has a test dataset, use the code again to create the test dataset.

9. If you're ready, train the model by following the instructions at Training your model.

Viewing your models

A project can have multiple versions of a model. You can use the console to view the models in a project. You can also use the `ListModels` operation.

> **Note**
>
> The list of models is eventually consistent. If you create a model, you might have to wait a short while before the models list is up to date.

Viewing your models (console)

Perform the steps in the following procedure to view your project's models in the console.

To view your models (console)

1. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
2. Choose **Get started**.
3. In the left navigation pane, choose **Projects**.
4. On the **Projects** page, select the project that contains the models that you want to view.
5. In the left navigation pane, choose **Models** and then view the model details.

Viewing your models (SDK)

To get view the versions of a model you use the `ListModels` operation. To get information about a specific model version, use the `DescribeModel` operation. The following example lists all the model versions in a project and then displays performance and output configuration information for individual model versions.

To view your models (SDK)

1. If you haven't already done so, install and configure the AWS CLI and the AWS SDKs. For more information, see Step 4: Set up the AWS CLI and AWS SDKs.
2. Use the following example code to list your models and get information about a model.

   CLI

   Use the `list-models` command to list the models in a project.

   Change the following value:

- `project-name` to the name of the project that contains the model that you want to view.

```
aws lookoutvision list-models --project-name project name \
  --profile lookoutvision-access
```

Use the `describe-model` command to get information about a model. Change the following values:

- `project-name` to the name of the project that contains the model that you want to view.
- `model-version` to the version of the model that you want to describe.

```
aws lookoutvision describe-model --project-name project name\
    --model-version model version \
   --profile lookoutvision-access
```

Python

This code is taken from the AWS Documentation SDK examples GitHub repository. See the full example here.

```
    @staticmethod
    def describe_models(lookoutvision_client, project_name):
        """
        Gets information about all models in a Lookout for Vision project.

        :param lookoutvision_client: A Boto3 Lookout for Vision client.
        :param project_name: The name of the project that you want to use.
        """
        try:
            response =
 lookoutvision_client.list_models(ProjectName=project_name)
            print("Project: " + project_name)
            for model in response["Models"]:
                Models.describe_model(
                    lookoutvision_client, project_name, model["ModelVersion"]
                )
```

```
                print()
            print("Done...")
        except ClientError:
            logger.exception("Couldn't list models.")
            raise
```

Java V2

This code is taken from the AWS Documentation SDK examples GitHub repository. See the full example here.

```
/**
 * Lists the models in an Amazon Lookout for Vision project.
 *
 * @param lfvClient   An Amazon Lookout for Vision client.
 * @param projectName The name of the project that contains the models that
 *                    you want to list.
 * @return List <Metadata> A list of models in the project.
 */
public static List<ModelMetadata> listModels(LookoutVisionClient lfvClient,
 String projectName)
                throws LookoutVisionException {

        ListModelsRequest listModelsRequest = ListModelsRequest.builder()
                        .projectName(projectName)
                        .build();

        // Get a list of models in the supplied project.
        ListModelsResponse response = lfvClient.listModels(listModelsRequest);

        for (ModelMetadata model : response.models()) {
                logger.log(Level.INFO, "Model ARN: {0}\nVersion: {1}\nStatus:
 {2}\nMessage: {3}", new Object[] {
                                model.modelArn(),
                                model.modelVersion(),
                                model.statusMessage(),
                                model.statusAsString() });
        }

        return response.models();
```

```
}
```

Deleting a model

You can delete a version of a model by using the console or by using the `DeleteModel` operation. You can't delete model version that is running or being trained.

If the model is version running, first use the `StopModel` operation to stop the model version. For more information, see Stopping your Amazon Lookout for Vision model. If a model is training, wait until it finishes before you delete the model.

It might take a few seconds to delete a model. To determine if a model has been deleted, call ListProjects and check if the version of the model (`ModelVersion`) is in the `Models` array.

Deleting a model (console)

Perform the following steps to delete a model from the console.

To delete a model (console)

1. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
2. Choose **Get started**.
3. In the left navigation pane, choose **Projects**.
4. On the **Projects** page, select the project that contains the model that you want to delete.
5. In the left navigation pane, choose **Models**.
6. On the **models** view, select the radio button for the model that you want to delete.
7. Choose **Delete** at the top of the page.
8. In the **Delete** dialog box, enter **delete** to confirm that you want to delete the model.
9. Choose **Delete model** to delete the model.

Deleting a model (SDK)

Use the following procedure to delete model with the `DeleteModel` operation.

To delete a model (SDK)

1. If you haven't already done so, install and configure the AWS CLI and the AWS SDKs. For more information, see Step 4: Set up the AWS CLI and AWS SDKs.
2. Use the following example code to delete a model.

   CLI

   Change the following values:

   - `project-name` to the name of the project that contains the model that you want to delete.
   - `model-version` to the version of the model that you want to delete.

```
aws lookoutvision delete-model --project-name project name\
  --model-version model version \
  --profile lookoutvision-access
```

   Python

   This code is taken from the AWS Documentation SDK examples GitHub repository. See the full example here.

```
    @staticmethod
    def delete_model(lookoutvision_client, project_name, model_version):
        """
        Deletes a Lookout for Vision model. The model must first be stopped and
 can't
        be in training.

        :param lookoutvision_client: A Boto3 Lookout for Vision client.
        :param project_name: The name of the project that contains the desired
 model.
        :param model_version: The version of the model that you want to delete.
        """
        try:
            logger.info("Deleting model: %s", model_version)
            lookoutvision_client.delete_model(
                ProjectName=project_name, ModelVersion=model_version
            )
```

```
            model_exists = True
            while model_exists:
                response =
 lookoutvision_client.list_models(ProjectName=project_name)

                model_exists = False
                for model in response["Models"]:
                    if model["ModelVersion"] == model_version:
                        model_exists = True

                if model_exists is False:
                    logger.info("Model deleted")
                else:
                    logger.info("Model is being deleted...")
                    time.sleep(2)

            logger.info("Deleted Model: %s", model_version)
        except ClientError:
            logger.exception("Couldn't delete model.")
            raise
```

Java V2

This code is taken from the AWS Documentation SDK examples GitHub repository. See the full example here.

```
/**
* Deletes an Amazon Lookout for Vision model.
*
* @param lfvClient    An Amazon Lookout for Vision client. Returns after the
 model is deleted.
* @param projectName  The name of the project that contains the model that you
 want to delete.
* @param modelVersion The version of the model that you want to delete.
* @return void
*/
public static void deleteModel(LookoutVisionClient lfvClient,
                String projectName,
                String modelVersion) throws LookoutVisionException,
 InterruptedException {
```

```
        DeleteModelRequest deleteModelRequest = DeleteModelRequest.builder()
                        .projectName(projectName)
                        .modelVersion(modelVersion)
                        .build();

        lfvClient.deleteModel(deleteModelRequest);

        boolean deleted = false;

        do {

                ListModelsRequest listModelsRequest =
ListModelsRequest.builder()
                                .projectName(projectName)
                                .build();

                // Get a list of models in the supplied project.
                ListModelsResponse response =
lfvClient.listModels(listModelsRequest);

                ModelMetadata modelMetadata = response.models().stream()
                                .filter(model ->
model.modelVersion().equals(modelVersion)).findFirst()
                                .orElse(null);

                if (modelMetadata == null) {
                        deleted = true;
                        logger.log(Level.INFO, "Deleted: Model version {0} of
project {1}.",
                                        new Object[] { modelVersion,
projectName });

                } else {
                        logger.log(Level.INFO, "Not yet deleted: Model version
{0} of project {1}.",
                                        new Object[] { modelVersion,
projectName });
                        TimeUnit.SECONDS.sleep(60);
                }

        } while (!deleted);
```

```
}
```

Tagging models

You can identify, organize, search for, and filter your Amazon Lookout for Vision models by using tags. Each tag is a label consisting of a user-defined key and value. For example, to help determine billing for your models, you could tag your models with a `Cost center` key and add the appropriate cost center number as a value. For more information, see Tagging AWS resources.

Use tags to:

- Track billing for a model by using cost allocation tags. For more information, see Using Cost Allocation Tags.
- Control access to a model by using Identity and Access Management (IAM). For more information, see Controlling access to AWS resources using resource tags.
- Automate model management. For example, you can run automated start or stop scripts that turn off development models during non-business hours to reduce costs. For more information, see Running your trained Amazon Lookout for Vision model.

You can tag models by using the Amazon Lookout for Vision console or by using the AWS SDKs.

Topics

- Tagging models (console)
- Tagging models (SDK)

Tagging models (console)

You can use the Amazon Lookout for Vision console to add tags to models, view the tags attached to a model, and remove tags.

Adding or removing tags (console)

This procedure explains how to add tags to, or remove tags from, an existing model. You can also add tags to a new model when it is trained. For more information, see Training your model.

To add tags to, or remove tags from, an existing model (console)

1. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
2. Choose **Get started**.
3. In the navigation pane, choose **Projects**.
4. On the **Projects** resources page, choose the project that contains the model that you want to tag.
5. In the navigation pane, under the project you previously chose, choose **Models**.
6. In the **Models** section, choose the model that you want to add a tag to.
7. On the model's details page, choose the **Tags** tab.
8. In the **Tags** section, choose **Manage tags**.
9. On the **Manage tags** page, choose **Add new tag**.
10. Enter a key and a value.
    a. For **Key**, enter a name for the key.
    b. For **Value**, enter a value.
11. To add more tags, repeat steps 9 and 10.
12. (Optional) To remove a tag, choose **Remove** next to the tag that you want to remove. If you are removing a previously saved tag, it is removed when you save your changes.
13. Choose **Save changes** to save your changes.

Viewing model tags (console)

You can use the Amazon Lookout for Vision console to view the tags attached to a model.

To view the tags attached to *all models within a project*, you must use the AWS SDK. For more information, see Listing model tags (SDK).

To view the tags attached to a model

1. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
2. Choose **Get started**.
3. In the navigation pane, choose **Projects**.

4. On the **Projects** resources page, choose the project that contains the model whose tag you want to view.
5. In the navigation pane, under the project you previously chose, choose **Models**.
6. In the **Models** section, choose the model whose tag you want to view.
7. On the model's details page, choose the **Tags** tab. The tags are shown in **Tags** section.

Tagging models (SDK)

You can use the AWS SDK to:

- Add tags to a new model
- Add tags to an existing model
- List the tags attached to a model
- Remove tags from a model

This section includes AWS CLI examples. If you haven't installed the AWS CLI, see Step 4: Set up the AWS CLI and AWS SDKs.

Adding tags to a new model (SDK)

You can add tags to a model when you create it using the CreateModel operation. Specify one or more tags in the `Tags` array input parameter.

```
aws lookoutvision create-model --project-name "project name"\
  --output-config '{ "S3Location": { "Bucket": "output bucket", "Prefix":  "output
 folder" } }'\
  --tags '[{"Key":"Key","Value":"Value"}]' \
  --profile lookoutvision-access
```

For information about creating and training a model, see Training a model (SDK).

Adding tags to an existing model (SDK)

To add one or more tags to an existing model, use the TagResource operation. Specify the model's Amazon Resource Name (ARN) (`ResourceArn`) and the tags (`Tags`) that you want to add.

```
aws lookoutvision tag-resource --resource-arn "resource-arn"\
```

```
  --tags '[{"Key":"Key","Value":"Value"}]' \
  --profile lookoutvision-access
```

For example Java code, see TagModel.

Listing model tags (SDK)

To list the tags attached to a model, use the ListTagsForResource operation and specify the model's Amazon Resource Name (ARN), the (`ResourceArn`). The response is a map of tag keys and values that are attached to the specified model.

```
aws lookoutvision list-tags-for-resource --resource-arn resource-arn \
  --profile lookoutvision-access
```

To see which models in a project have a specific tag, call `ListModels` to get a list of models. Then call `ListTagsForResource` for each model in the response from `ListModels`. Inspect the response from `ListTagsForResource` to see if the required tag is present.

For example Java code, see ListModelTags. For example Python code that searches for a tag value across all projects, see find_tag.py.

Removing tags from a model (SDK)

To remove one or more tags from a model, use the UntagResource operation. Specify the model's Amazon Resource Name (ARN) (`ResourceArn`) and the tag keys (`Tag-Keys`) that you want to remove.

```
aws lookoutvision untag-resource --resource-arn resource-arn\
  --tag-keys '["Key"]' \
  --profile lookoutvision-access
```

For example Java code, see UntagModel.

Viewing your trial detection tasks

You can view your trial detections by using the console. You can't use the AWS SDK to view trial detection tasks.

> **Note**
>
> The list of trial detections is eventually consistent. If you create a trial detection, you might have to wait a short while before the trial detections list is up to date.

Viewing your trial detection tasks (console)

Use the following procedures to view your trial detections.

To view your trial detection tasks

1. Open the Amazon Lookout for Vision console at https://console.aws.amazon.com/lookoutvision/.
2. Choose **Get started**.
3. In the left navigation pane, choose **Trial detections**.
4. On the trial detections page, choose a trial detection task to view its details.

Example code and datasets

The following are code examples and datasets that you can use with Amazon Lookout for Vision.

Topics

- Example code
- Example datasets

Example code

The following code examples for Amazon Lookout for Vision are available.

Example	Description
GitHub	Example Python code that trains and hosts an Amazon Lookout for Vision model.
Amazon Lookout for Vision Lab	A Python Notebook that you can use to create a model with the circuitboard example images.
Python example code	Python examples used in the Amazon Lookout for Vision documentation.
Java example code	Java examples used in the Amazon Lookout for Vision documentation.

Example datasets

The following are example datasets that you can use with Amazon Lookout for Vision.

Topics

- Image segmentation datasets
- Image classification dataset

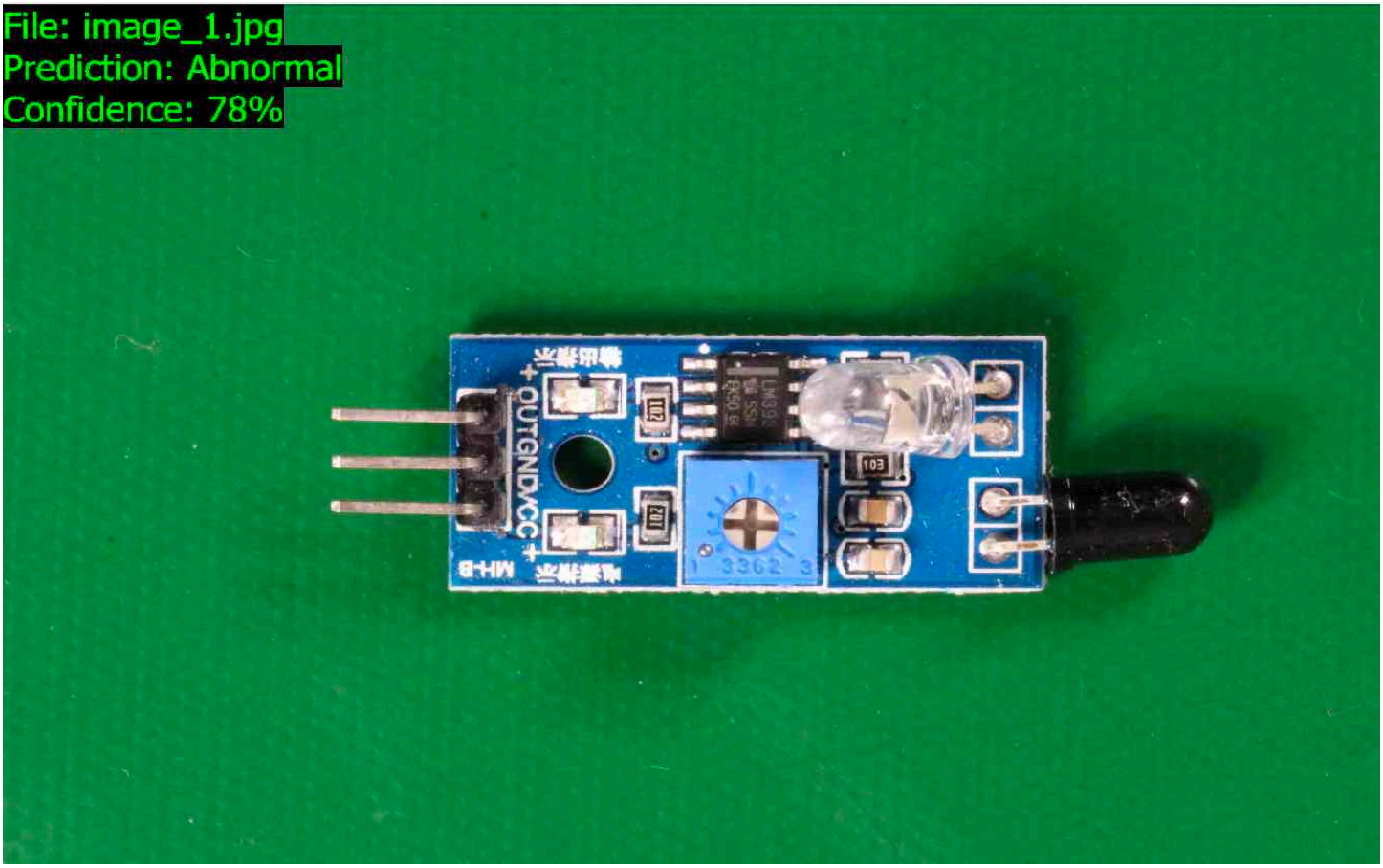
Image segmentation datasets

Getting started with Amazon Lookout for Vision provides a dataset of broken cookies that you can use to create an image segmentation model.

For another dataset that creates an image segmentation model, see Identify the location of anomalies using Amazon Lookout for Vision at the edge without using a GPU.

Image classification dataset

Amazon Lookout for Vision provides example images of circuit boards that you can use to create an image classification model.



You can copy the images from the https://github.com/aws-samples/amazon-lookout-for-vision GitHub repository. The images are in the `circuitboard` folder.

The `circuitboard` folder has the following folders.

- `train` – Images you can use in a training dataset.

- `test` – Images you can use in a test dataset.
- `extra_images` – Images you can use to run a trial detection or to try out your trained model with the DetectAnomalies operation.

The `train` and `test` folders each have a subfolder named `normal` (contains images that are normal) and a subfolder named `anomaly` (contains images with anomalies).

> **Note**
>
> Later, when you create a dataset with the console, Amazon Lookout for Vision can use the folder names (`normal` and `anomaly`) to label the images automatically. For more information, see the section called "Amazon S3 bucket".

To prepare the dataset images

1. Clone the https://github.com/aws-samples/amazon-lookout-for-vision repository to your computer. For more information, see Cloning a repository.
2. Create an Amazon S3 bucket. For more information, see How do I create an S3 Bucket?.
3. At the command prompt, enter the following command to copy the dataset images from your computer to your Amazon S3 bucket.

```
aws s3 cp --recursive your-repository-folder/circuitboard s3://your-bucket/
circuitboard
```

After uploading the images, you can create a model. You can automatically classify the images by adding the images from the Amazon S3 location that you previously uploaded the circuit board images to. Remember that you are charged for each successful training of a model and for the amount of time that a model is running (hosted).

To create a classification model

1. Do Creating a project (console).
2. Do Creating a dataset using images stored in an Amazon S3 bucket.
   - For step 6, choose the **Separate training and test datasets** tab.

- For step 8a, enter the S3 URI for the training images you uploaded in To prepare the dataset images. For example `s3://your-bucket/circuitboard/train`. For step 8b, enter the S3 URI for the test dataset. For example, `s3://your-bucket/circuitboard/test`.
- Be sure to do step 9.

3. Do Training a model (console).
4. Do Starting your model (console).
5. Do Detecting anomalies in an image. You can use images from the `test_images` folder.
6. When you're finished with the model, do Stopping your model (console).

Security in Amazon Lookout for Vision

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from data centers and network architectures that are built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS Compliance Programs. To learn about the compliance programs that apply to Amazon Lookout for Vision, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using Lookout for Vision. The following topics show you how to configure Lookout for Vision to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your Lookout for Vision resources.

Topics

- Data protection in Amazon Lookout for Vision
- Identity and access management for Amazon Lookout for Vision
- Compliance validation for Amazon Lookout for Vision
- Resilience in Amazon Lookout for Vision
- Infrastructure security in Amazon Lookout for Vision

Data protection in Amazon Lookout for Vision

The AWS shared responsibility model applies to data protection in Amazon Lookout for Vision. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on

this infrastructure. You are also responsible for the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the Data Privacy FAQ. For information about data protection in Europe, see the AWS Shared Responsibility Model and GDPR blog post on the *AWS Security Blog*.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail. For information about using CloudTrail trails to capture AWS activities, see Working with CloudTrail trails in the *AWS CloudTrail User Guide*.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.
- If you require FIPS 140-3 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-3.

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with Lookout for Vision or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

Data encryption

The following information explains where Amazon Lookout for Vision uses data encryption to protect your data.

Encryption at rest

Images

To train your model, Amazon Lookout for Vision makes a copy of your source training and test images. The copied images are encrypted at rest in Amazon Simple Storage Service (S3) using server-side encryption with an AWS owned key or a key that you provide. The keys are stored using AWS Key Management Service (SSE-KMS). Your source images are unaffected. For more information, see Training your model.

Amazon Lookout for Vision models

By default, trained models and manifest files are encrypted in Amazon S3 using server-side encryption with KMS keys stored in AWS Key Management Service (SSE-KMS). Lookout for Vision uses an AWS owned key. For more information, see Protecting Data Using Server-Side Encryption. Training results are written to the bucket specified in the `output_bucket` input parameter to `CreateModel`. The training results are encrypted using the configured encryption settings for the bucket (`output_bucket`).

Amazon Lookout for Vision console bucket

The Amazon Lookout for Vision console creates an Amazon S3 bucket (console bucket) that you can use to manage your projects. The console bucket is encrypted using the default Amazon S3 encryption. For more information, see Amazon Simple Storage Service default encryption for S3 buckets. If you are using your own KMS key, configure the console bucket after it is created. For more information, see Protecting Data Using Server-Side Encryption. Amazon Lookout for Vision blocks public access to the console bucket.

Encryption in transit

Amazon Lookout for Vision API endpoints only support secure connections over HTTPS. All communication is encrypted with Transport Layer Security (TLS).

Key management

You can use AWS Key Management Service (KMS) to manage encryption for the input images that you store in Amazon S3 buckets. For more information, see Step 5: (Optional) Using your own AWS Key Management Service key.

By default your images are encrypted with a key that AWS owns and manages. You can also choose to use your own AWS Key Management Service (KMS) key. For more information, see AWS Key Management Service concepts.

Internetwork traffic privacy

An Amazon Virtual Private Cloud (Amazon VPC) endpoint for Amazon Lookout for Vision is a logical entity within a VPC that allows connectivity only to Amazon Lookout for Vision. Amazon VPC routes requests to Amazon Lookout for Vision and routes responses back to the VPC. For more information, see VPC Endpoints in the *Amazon VPC User Guide*. For information about using Amazon VPC endpoints with Amazon Lookout for Vision see Access Amazon Lookout for Vision using an interface endpoint (AWS PrivateLink).

Identity and access management for Amazon Lookout for Vision

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use Lookout for Vision resources. IAM is an AWS service that you can use with no additional charge.

Topics

- Audience
- Authenticating with identities
- Managing access using policies
- How Amazon Lookout for Vision works with IAM
- Amazon Lookout for Vision identity-based policy examples
- AWS managed policies for Amazon Lookout for Vision
- Troubleshooting Amazon Lookout for Vision identity and access

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in Lookout for Vision.

Service user – If you use the Lookout for Vision service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more Lookout for Vision features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in Lookout for Vision, see Troubleshooting Amazon Lookout for Vision identity and access.

Service administrator – If you're in charge of Lookout for Vision resources at your company, you probably have full access to Lookout for Vision. It's your job to determine which Lookout for Vision features and resources your service users should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with Lookout for Vision, see How Amazon Lookout for Vision works with IAM.

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to Lookout for Vision. To view example Lookout for Vision identity-based policies that you can use in IAM, see Amazon Lookout for Vision identity-based policy examples.

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be *authenticated* (signed in to AWS) as the AWS account root user, as an IAM user, or by assuming an IAM role.

You can sign in to AWS as a federated identity by using credentials provided through an identity source. AWS IAM Identity Center (IAM Identity Center) users, your company's single sign-on authentication, and your Google or Facebook credentials are examples of federated identities. When you sign in as a federated identity, your administrator previously set up identity federation using IAM roles. When you access AWS by using federation, you are indirectly assuming a role.

Depending on the type of user you are, you can sign in to the AWS Management Console or the AWS access portal. For more information about signing in to AWS, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

If you access AWS programmatically, AWS provides a software development kit (SDK) and a command line interface (CLI) to cryptographically sign your requests by using your credentials. If you don't use AWS tools, you must sign requests yourself. For more information about using the

recommended method to sign requests yourself, see AWS Signature Version 4 for API requests in the *IAM User Guide*.

Regardless of the authentication method that you use, you might be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see Multi-factor authentication in the *AWS IAM Identity Center User Guide* and AWS Multi-factor authentication in IAM in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account *root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you don't use the root user for your everyday tasks. Safeguard your root user credentials and use them to perform the tasks that only the root user can perform. For the complete list of tasks that require you to sign in as the root user, see Tasks that require root user credentials in the *IAM User Guide*.

Federated identity

As a best practice, require human users, including users that require administrator access, to use federation with an identity provider to access AWS services by using temporary credentials.

A *federated identity* is a user from your enterprise user directory, a web identity provider, the AWS Directory Service, the Identity Center directory, or any user that accesses AWS services by using credentials provided through an identity source. When federated identities access AWS accounts, they assume roles, and the roles provide temporary credentials.

For centralized access management, we recommend that you use AWS IAM Identity Center. You can create users and groups in IAM Identity Center, or you can connect and synchronize to a set of users and groups in your own identity source for use across all your AWS accounts and applications. For information about IAM Identity Center, see What is IAM Identity Center? in the *AWS IAM Identity Center User Guide*.

IAM users and groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. Where possible, we recommend relying on temporary credentials instead of creating

IAM users who have long-term credentials such as passwords and access keys. However, if you have specific use cases that require long-term credentials with IAM users, we recommend that you rotate access keys. For more information, see Rotate access keys regularly for use cases that require long-term credentials in the *IAM User Guide*.

An *IAM group* is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see Use cases for IAM users in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. To temporarily assume an IAM role in the AWS Management Console, you can switch from a user to an IAM role (console). You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see Methods to assume a role in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Federated user access** – To assign permissions to a federated identity, you create a role and define permissions for the role. When a federated identity authenticates, the identity is associated with the role and is granted the permissions that are defined by the role. For information about roles for federation, see Create a role for a third-party identity provider (federation) in the *IAM User Guide*. If you use IAM Identity Center, you configure a permission set. To control what your identities can access after they authenticate, IAM Identity Center correlates the permission set to a role in IAM. For information about permissions sets, see Permission sets in the *AWS IAM Identity Center User Guide*.
- **Temporary IAM user permissions** – An IAM user or role can assume an IAM role to temporarily take on different permissions for a specific task.
- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource

(instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.

- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.
  - **Forward access sessions (FAS)** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see Forward access sessions.
  - **Service role** – A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Create a role to delegate permissions to an AWS service in the *IAM User Guide*.
  - **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.
- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see Use an IAM role to grant permissions to applications running on Amazon EC2 instances in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy is an object in AWS that, when associated with an identity or resource, defines their

permissions. AWS evaluates these policies when a principal (user, root user, or role session) makes a request. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

Identity-based policies can be further categorized as *inline policies* or *managed policies*. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see Choose between managed policies and inline policies in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Access control lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see Access control list (ACL) overview in the *Amazon Simple Storage Service Developer Guide*.

Other policy types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of an entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the `Principal` field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see Permissions boundaries for IAM entities in the *IAM User Guide*.
- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see Service control policies in the *AWS Organizations User Guide*.
- **Resource control policies (RCPs)** – RCPs are JSON policies that you can use to set the maximum available permissions for resources in your accounts without updating the IAM policies attached to each resource that you own. The RCP limits permissions for resources in member accounts and can impact the effective permissions for identities, including the AWS account root user, regardless of whether they belong to your organization. For more information about

Organizations and RCPs, including a list of AWS services that support RCPs, see Resource control policies (RCPs) in the *AWS Organizations User Guide*.

- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see Session policies in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the *IAM User Guide*.

How Amazon Lookout for Vision works with IAM

Before you use IAM to manage access to Lookout for Vision, learn what IAM features are available to use with Lookout for Vision.

IAM features you can use with Amazon Lookout for Vision

IAM feature	Lookout for Vision support
Identity-based policies	Yes
Resource-based policies	No
Policy actions	Yes
Policy resources	Yes
Policy condition keys (service-specific)	Yes
ACLs	No
ABAC (tags in policies)	Partial
Temporary credentials	Yes

IAM feature	Lookout for Vision support
Forward access sessions (FAS)	Yes
Service roles	No
Service-linked roles	No

To get a high-level view of how Lookout for Vision and other AWS services work with most IAM features, see AWS services that work with IAM in the *IAM User Guide*.

Identity-based policies for Lookout for Vision

Supports identity-based policies: Yes

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. You can't specify the principal in an identity-based policy because it applies to the user or role to which it is attached. To learn about all of the elements that you can use in a JSON policy, see IAM JSON policy elements reference in the *IAM User Guide*.

Identity-based policy examples for Lookout for Vision

To view examples of Lookout for Vision identity-based policies, see Amazon Lookout for Vision identity-based policy examples.

Resource-based policies within Lookout for Vision

Supports resource-based policies: No

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified

principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

To enable cross-account access, you can specify an entire account or IAM entities in another account as the principal in a resource-based policy. Adding a cross-account principal to a resource-based policy is only half of establishing the trust relationship. When the principal and the resource are in different AWS accounts, an IAM administrator in the trusted account must also grant the principal entity (user or role) permission to access the resource. They grant permission by attaching an identity-based policy to the entity. However, if a resource-based policy grants access to a principal in the same account, no additional identity-based policy is required. For more information, see Cross account resource access in IAM in the *IAM User Guide*.

Policy actions for Lookout for Vision

Supports policy actions: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Policy actions usually have the same name as the associated AWS API operation. There are some exceptions, such as *permission-only actions* that don't have a matching API operation. There are also some operations that require multiple actions in a policy. These additional actions are called *dependent actions*.

Include actions in a policy to grant permissions to perform the associated operation.

To see a list of Lookout for Vision actions, see Actions defined by Amazon Lookout for Vision in the *Service Authorization Reference*.

Policy actions in Lookout for Vision use the following prefix before the action:

```
lookoutvision
```

To specify multiple actions in a single statement, separate them with commas.

```
"Action": [
      "lookoutvision:action1",
```

```
      "lookoutvision:action2"
         ]
```

To view examples of Lookout for Vision identity-based policies, see Amazon Lookout for Vision identity-based policy examples.

Policy resources for Lookout for Vision

Supports policy resources: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Resource` JSON policy element specifies the object or objects to which the action applies. Statements must include either a `Resource` or a `NotResource` element. As a best practice, specify a resource using its Amazon Resource Name (ARN). You can do this for actions that support a specific resource type, known as *resource-level permissions*.

For actions that don't support resource-level permissions, such as listing operations, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

To see a list of Lookout for Vision resource types and their ARNs, see Resources defined by Amazon Lookout for Vision in the *Service Authorization Reference*. To learn with which actions you can specify the ARN of each resource, see Actions defined by Amazon Lookout for Vision.

To view examples of Lookout for Vision identity-based policies, see Amazon Lookout for Vision identity-based policy examples.

Policy condition keys for Lookout for Vision

Supports service-specific policy condition keys: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Condition` element (or `Condition` *block*) lets you specify conditions in which a statement is in effect. The `Condition` element is optional. You can create conditional expressions that use condition operators, such as equals or less than, to match the condition in the policy with values in the request.

If you specify multiple `Condition` elements in a statement, or multiple keys in a single `Condition` element, AWS evaluates them using a logical `AND` operation. If you specify multiple values for a single condition key, AWS evaluates the condition using a logical `OR` operation. All of the conditions must be met before the statement's permissions are granted.

You can also use placeholder variables when you specify conditions. For example, you can grant an IAM user permission to access a resource only if it is tagged with their IAM user name. For more information, see IAM policy elements: variables and tags in the *IAM User Guide*.

AWS supports global condition keys and service-specific condition keys. To see all AWS global condition keys, see AWS global condition context keys in the *IAM User Guide*.

To see a list of Lookout for Vision condition keys, see Condition keys for Amazon Lookout for Vision in the *Service Authorization Reference*. To learn with which actions and resources you can use a condition key, see Actions defined by Amazon Lookout for Vision.

To view examples of Lookout for Vision identity-based policies, see Amazon Lookout for Vision identity-based policy examples.

ACLs in Lookout for Vision

Supports ACLs: No

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

ABAC with Lookout for Vision

Supports ABAC (tags in policies): Partial

Attribute-based access control (ABAC) is an authorization strategy that defines permissions based on attributes. In AWS, these attributes are called *tags*. You can attach tags to IAM entities (users or roles) and to many AWS resources. Tagging entities and resources is the first step of ABAC. Then

you design ABAC policies to allow operations when the principal's tag matches the tag on the resource that they are trying to access.

ABAC is helpful in environments that are growing rapidly and helps with situations where policy management becomes cumbersome.

To control access based on tags, you provide tag information in the condition element of a policy using the `aws:ResourceTag/key-name`, `aws:RequestTag/key-name`, or `aws:TagKeys` condition keys.

If a service supports all three condition keys for every resource type, then the value is **Yes** for the service. If a service supports all three condition keys for only some resource types, then the value is **Partial**.

For more information about ABAC, see Define permissions with ABAC authorization in the *IAM User Guide*. To view a tutorial with steps for setting up ABAC, see Use attribute-based access control (ABAC) in the *IAM User Guide*.

Using temporary credentials with Lookout for Vision

Supports temporary credentials: Yes

Some AWS services don't work when you sign in using temporary credentials. For additional information, including which AWS services work with temporary credentials, see AWS services that work with IAM in the *IAM User Guide*.

You are using temporary credentials if you sign in to the AWS Management Console using any method except a user name and password. For example, when you access AWS using your company's single sign-on (SSO) link, that process automatically creates temporary credentials. You also automatically create temporary credentials when you sign in to the console as a user and then switch roles. For more information about switching roles, see Switch from a user to an IAM role (console) in the *IAM User Guide*.

You can manually create temporary credentials using the AWS CLI or AWS API. You can then use those temporary credentials to access AWS. AWS recommends that you dynamically generate temporary credentials instead of using long-term access keys. For more information, see Temporary security credentials in IAM.

Forward access sessions for Lookout for Vision

Supports forward access sessions (FAS): Yes

When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see Forward access sessions.

Service roles for Lookout for Vision

Supports service roles: No

A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Create a role to delegate permissions to an AWS service in the *IAM User Guide*.

> **Warning**
>
> Changing the permissions for a service role might break Lookout for Vision functionality. Edit service roles only when Lookout for Vision provides guidance to do so.

Service-linked roles for Lookout for Vision

Supports service-linked roles: No

A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

For details about creating or managing service-linked roles, see AWS services that work with IAM. Find a service in the table that includes a `Yes` in the **Service-linked role** column. Choose the **Yes** link to view the service-linked role documentation for that service.

Amazon Lookout for Vision identity-based policy examples

By default, users and roles don't have permission to create or modify Lookout for Vision resources. They also can't perform tasks by using the AWS Management Console, AWS Command Line Interface (AWS CLI), or AWS API. To grant users permission to perform actions on the resources

that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

To learn how to create an IAM identity-based policy by using these example JSON policy documents, see Create IAM policies (console) in the *IAM User Guide*.

For details about actions and resource types defined by Lookout for Vision, including the format of the ARNs for each of the resource types, see Actions, resources, and condition keys for Amazon Lookout for Vision in the *Service Authorization Reference*.

Topics

- Policy best practices
- Accessing a single Amazon Lookout for Vision project
- Tag-based policy examples

Policy best practices

Identity-based policies determine whether someone can create, access, or delete Lookout for Vision resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see AWS managed policies or AWS managed policies for job functions in the *IAM User Guide*.
- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see Policies and permissions in IAM in the *IAM User Guide*.
- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as AWS CloudFormation. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.

- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see Validate policies with IAM Access Analyzer in the *IAM User Guide*.
- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see Secure API access with MFA in the *IAM User Guide*.

For more information about best practices in IAM, see Security best practices in IAM in the *IAM User Guide*.

Accessing a single Amazon Lookout for Vision project

In this example, you want to grant a user in your AWS account access to one of your Amazon Lookout for Vision projects.

```
{
    "Sid": "SpecificProjectOnly",
    "Effect": "Allow",
    "Action": [
        "lookoutvision:DetectAnomalies"
    ],
    "Resource": "arn:aws:lookoutvision:us-east-1:123456789012:model/myproject/*"
}
```

Tag-based policy examples

Tag-based policies are JSON policy documents that specify the actions that a principal can perform on tagged resources.

Use a tag to access a resource

This example policy grants a user or role in your AWS account permission to use the `DetectAnomalies` operation with any model tagged with the key `stage` and the value `production`.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "LookoutVision:DetectAnomalies"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/stage": "production"
                }
            }
        }
   ]
}
```

Use a tag to deny access to specific Amazon Lookout for Vision operations

This example policy denies permission for a user or role in your AWS account to call the `DeleteModel` or `StopModel` operations with any model tagged with the key `stage` and the value `production`.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "LookoutVision:DeleteModel",
                "LookoutVision:StopModel"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/stage": "production"
                }
            }
        }
   ]
}
```

AWS managed policies for Amazon Lookout for Vision

An AWS managed policy is a standalone policy that is created and administered by AWS. AWS managed policies are designed to provide permissions for many common use cases so that you can start assigning permissions to users, groups, and roles.

Keep in mind that AWS managed policies might not grant least-privilege permissions for your specific use cases because they're available for all AWS customers to use. We recommend that you reduce permissions further by defining customer managed policies that are specific to your use cases.

You cannot change the permissions defined in AWS managed policies. If AWS updates the permissions defined in an AWS managed policy, the update affects all principal identities (users, groups, and roles) that the policy is attached to. AWS is most likely to update an AWS managed policy when a new AWS service is launched or new API operations become available for existing services.

For more information, see AWS managed policies in the *IAM User Guide*.

AWS managed policy: AmazonLookoutVisionReadOnlyAccess

Use the `AmazonLookoutVisionReadOnlyAccess` policy to allow users read-only access to Amazon Lookout for Vision (and its dependencies) with the following Amazon Lookout for Vision actions (SDK operations). For example, you can use `DescribeModel` to get information about an existing model.

- DescribeDataset
- DescribeModel
- DescribeModelPackagingJob
- DescribeProject
- ListDatasetEntries
- ListModelPackagingJobs

- ListModels
- ListProjects
- ListTagsForResource

To call read-only actions, users don't need Amazon S3 bucket permissions. However, operation responses might include references to Amazon S3 buckets. For example, the `source-ref` entry in the response from `ListDatasetEntries` is a reference to an image in an Amazon S3 bucket. Add Amazon S3 bucket permissions if your users need to access referenced buckets. For example, a user might want to download an image referenced by a `source-ref` field. For more information, see Granting Amazon S3 Bucket permissions.

You can attach the `AmazonLookoutVisionReadOnlyAccess` policy to your IAM identities.

Permissions details

This policy includes the following permissions.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "LookoutVisionReadOnlyAccess",
            "Effect": "Allow",
            "Action": [
                "lookoutvision:DescribeDataset",
                "lookoutvision:DescribeModel",
                "lookoutvision:DescribeProject",
                "lookoutvision:DescribeModelPackagingJob",
                "lookoutvision:ListDatasetEntries",
                "lookoutvision:ListModels",
                "lookoutvision:ListProjects",
                "lookoutvision:ListTagsForResource",
                "lookoutvision:ListModelPackagingJobs"
            ],
            "Resource": "*"
        }
    ]
```

```
}
```

AWS managed policy: AmazonLookoutVisionFullAccess

Use the `AmazonLookoutVisionFullAccess` policy to allow users full access to Amazon Lookout for Vision (and its dependencies) with Amazon Lookout for Vision actions (SDK operations). For example, you can train a model without having to use the Amazon Lookout for Vision console. For more information, see Actions.

To create a dataset (`CreateDataset`) or create a model (`CreateModel`), your users must have full access permissions to the Amazon S3 bucket that stores dataset images, Amazon SageMaker AI Ground Truth manifest files, and training output. For more information, see Step 2: Set up permissions.

You can also give permission to Amazon Lookout for Vision SDK actions by using the `AmazonLookoutVisionConsoleFullAccess` policy.

You can attach the `AmazonLookoutVisionFullAccess` policy to your IAM identities.

Permissions details

This policy includes the following permissions.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "LookoutVisionFullAccess",
            "Effect": "Allow",
            "Action": [
                "lookoutvision:*"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS managed policy: AmazonLookoutVisionConsoleFullAccess

Use the `AmazonLookoutVisionFullAccess` policy to allow users full access to the Amazon Lookout for Vision console, actions (SDK operations), and any dependencies that the service has. For more information, see Getting started with Amazon Lookout for Vision.

The `LookoutVisionConsoleFullAccess` policy includes permissions to your Amazon Lookout for Vision console bucket. For information about the console bucket, see Step 3: Create the console bucket. To store datasets, images, and Amazon SageMaker AI Ground Truth manifest files in a different Amazon S3 bucket, your users need additional permissions. For more information, see the section called "Setting Amazon S3 bucket permissions".

You can attach the `AmazonLookoutVisionConsoleFullAccess` policy to your IAM identities.

Permissions groupings

This policy is grouped into statements based on the set of permissions provided:

- `LookoutVisionFullAccess` – Allows access to perform all Lookout for Vision actions.
- `LookoutVisionConsoleS3BucketSearchAccess` – Allows listing of all Amazon S3 buckets owned by the caller. Lookout for Vision uses this action to identify the AWS Region-specific Lookout for Vision console bucket, if one exists in the caller's account.
- `LookoutVisionConsoleS3BucketFirstUseSetupAccessPermissions` – Allows creating and configuring Amazon S3 buckets that match the Lookout for Vision console bucket name pattern. Lookout for Vision uses these actions to create and configure a Region-specific Lookout for Vision console bucket when it can't find one.
- `LookoutVisionConsoleS3BucketAccess` – Allows dependent Amazon S3 actions on buckets that match the Lookout for Vision console bucket name pattern. Lookout for Vision uses `s3:ListBucket` to search for image objects when creating a dataset from an Amazon S3 bucket and when starting a trial detection task. Lookout for Vision uses `s3:GetBucketLocation` and `s3:GetBucketVersioning` to validate the bucket's AWS Region, owner, and configuration as part of the following:
  - Creating a dataset
  - Training a model
  - Starting a trial detection task
  - Performing trial detection feedback

`LookoutVisionConsoleS3ObjectAccess` – Allows reading and writing of Amazon S3 objects inside buckets that match the Lookout for Vision Console bucket name pattern. Lookout for Vision uses these actions to display images in console gallery views and to upload new images for use in datasets. Additionally, these permissions allow Lookout for Vision to write out metadata while creating a dataset, training a model, starting a trial detection task, and performing trial detection feedback.

- `LookoutVisionConsoleDatasetLabelingToolsAccess` – Allows dependent Amazon SageMaker AI GroundTruth labeling actions. Lookout for Vision uses these actions to scan S3 buckets for images, create GroundTruth manifest files, and to annotate trial detection task results with validation labels.
- `LookoutVisionConsoleDashboardAccess` - Allows reading of Amazon CloudWatch metrics. Lookout for Vision uses these actions to populate the dashboard graphs and anomalies-detected statistics.
- `LookoutVisionConsoleTagSelectorAccess` – Allows reading account-specific tag key and tag value suggestions. Lookout for Vision uses these permissions to provide recommendations for tag keys and tag values within the **Manage tags** console pages.
- `LookoutVisionConsoleKmsKeySelectorAccess` – Allows listing AWS Key Management Service (KMS) keys and aliases. Amazon Lookout for Vision uses this permission to populate the KMS keys in the suggested **Tags** selection on certain Lookout for Vision actions that support customer managed KMS keys for encryption.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "LookoutVisionFullAccess",
            "Effect": "Allow",
            "Action": [
                "lookoutvision:*"
            ],
            "Resource": "*"
        },
        {
            "Sid": "LookoutVisionConsoleS3BucketSearchAccess",
            "Effect": "Allow",
            "Action": [
                "s3:ListAllMyBuckets"
```

```
            ],
            "Resource": "*"
        },
        {
            "Sid": "LookoutVisionConsoleS3BucketFirstUseSetupAccess",
            "Effect": "Allow",
            "Action": [
                "s3:CreateBucket",
                "s3:PutBucketVersioning",
                "s3:PutLifecycleConfiguration",
                "s3:PutEncryptionConfiguration",
                "s3:PutBucketPublicAccessBlock"
            ],
            "Resource": "arn:aws:s3:::lookoutvision-*"
        },
        {
            "Sid": "LookoutVisionConsoleS3BucketAccess",
            "Effect": "Allow",
            "Action": [
                "s3:ListBucket",
                "s3:GetBucketLocation",
                "s3:GetBucketAcl",
                "s3:GetBucketVersioning"
            ],
            "Resource": "arn:aws:s3:::lookoutvision-*"
        },
        {
            "Sid": "LookoutVisionConsoleS3ObjectAccess",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:GetObjectVersion",
                "s3:PutObject",
                "s3:AbortMultipartUpload",
                "s3:ListMultipartUploadParts"
            ],
            "Resource": "arn:aws:s3:::lookoutvision-*/*"
        },
        {
            "Sid": "LookoutVisionConsoleDatasetLabelingToolsAccess",
            "Effect": "Allow",
            "Action": [
                "groundtruthlabeling:RunGenerateManifestByCrawlingJob",
                "groundtruthlabeling:AssociatePatchToManifestJob",
```

```
                "groundtruthlabeling:DescribeConsoleJob"
            ],
            "Resource": "*"
        },
        {
            "Sid": "LookoutVisionConsoleDashboardAccess",
            "Effect": "Allow",
            "Action": [
                "cloudwatch:GetMetricData",
                "cloudwatch:GetMetricStatistics"
            ],
            "Resource": "*"
        },
        {
            "Sid": "LookoutVisionConsoleTagSelectorAccess",
            "Effect": "Allow",
            "Action": [
                "tag:GetTagKeys",
                "tag:GetTagValues"
            ],
            "Resource": "*"
        },
        {
            "Sid": "LookoutVisionConsoleKmsKeySelectorAccess",
            "Effect": "Allow",
            "Action": [
                "kms:ListAliases"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS managed policy: AmazonLookoutVisionConsoleReadOnlyAccess

Use the `AmazonLookoutVisionConsoleReadOnlyAccess` policy to allow users read-only access to the Amazon Lookout for Vision console, actions (SDK operations), and any dependencies that the service has.

The `AmazonLookoutVisionConsoleReadOnlyAccess` policy includes Amazon S3 permissions for the Amazon Lookout for Vision console bucket. If your dataset images or Amazon SageMaker AI Ground Truth manifest files are in a different Amazon S3 bucket, your users need additional permissions. For more information, see the section called “Setting Amazon S3 bucket permissions”.

You can attach the `AmazonLookoutVisionConsoleReadOnlyAccess` policy to your IAM identities.

Permissions groupings

This policy is grouped into statements based on the set of permissions provided:

- `LookoutVisionReadOnlyAccess` – Allows access to perform read-only Lookout for Vision actions.
- `LookoutVisionConsoleS3BucketSearchAccess` – Allows listing of all S3 buckets owned by the caller. Lookout for Vision uses this action to identify the AWS Region-specific Lookout for Vision console bucket, if there is one in the caller's account.
- `LookoutVisionConsoleS3ObjectReadAccess` – Allows reading Amazon S3 objects and Amazon S3 object versions in Lookout for Vision console buckets. Lookout for Vision uses these actions to display the images in datasets, models, and trial detections.
- `LookoutVisionConsoleDashboardAccess` – Allows reading Amazon CloudWatch metrics. Lookout for Vision uses these actions to populate statistics for dashboard graphs and anomalies detected.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "LookoutVisionReadOnlyAccess",
            "Effect": "Allow",
            "Action": [
                "lookoutvision:DescribeDataset",
                "lookoutvision:DescribeModel",
                "lookoutvision:DescribeProject",
                "lookoutvision:DescribeTrialDetection",
                "lookoutvision:DescribeModelPackagingJob",
                "lookoutvision:ListDatasetEntries",
                "lookoutvision:ListModels",
                "lookoutvision:ListProjects",
                "lookoutvision:ListTagsForResource",
                "lookoutvision:ListTrialDetections",
                "lookoutvision:ListModelPackagingJobs"
```

```
            ],
            "Resource": "*"
        },
        {
            "Sid": "LookoutVisionConsoleS3BucketSearchAccess",
            "Effect": "Allow",
            "Action": [
                "s3:ListAllMyBuckets"
            ],
            "Resource": "*"
        },
        {
            "Sid": "LookoutVisionConsoleS3ObjectReadAccess",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:GetObjectVersion"
            ],
            "Resource": "arn:aws:s3:::lookoutvision-*/*"
        },
        {
            "Sid": "LookoutVisionConsoleDashboardAccess",
            "Effect": "Allow",
            "Action": [
                "cloudwatch:GetMetricData",
                "cloudwatch:GetMetricStatistics"
            ],
            "Resource": "*"
        }
    ]
}
```

Lookout for Vision updates to AWS managed policies

View details about updates to AWS managed policies for Lookout for Vision since this service began tracking these changes. For automatic alerts about changes to this page, subscribe to the RSS feed on the Lookout for Vision Document history page.

	packaging operations to the AmazonLookoutVisionFullAccess and AmazonLookoutVisionConsoleFullAccess policies: • DescribeModelPackagingJob • ListModelPackagingJobs • StartModelPackagingJob **Amazon Lookout for Vision added the following model packaging operations to the AmazonLookoutVisionReadOnlyAccess and AmazonLookoutVisionConsoleReadOnlyAccess policies:** • DescribeModelPackagingJob • ListModelPackagingJobs	
New policies added	**Amazon Lookout for Vision added the following policies.** • AmazonLookoutVisionReadOnlyAccess • AmazonLookoutVisionFullAccess • AmazonLookoutVisionConsoleFullAccess • AmazonLookoutVisionConsoleReadOnlyAccess	**May 11th, 2021**
Lookout for Vision started tracking changes	Amazon Lookout for Vision started tracking changes for its AWS managed policies.	March 1st, 2021

Troubleshooting Amazon Lookout for Vision identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with Lookout for Vision and IAM.

Topics

- I am not authorized to perform an action in Lookout for Vision
- I am not authorized to perform iam:PassRole
- I want to allow people outside of my AWS account to access my Lookout for Vision resources

I am not authorized to perform an action in Lookout for Vision

If you receive an error that you're not authorized to perform an action, your policies must be updated to allow you to perform the action.

The following example error occurs when the `mateojackson` IAM user tries to use the console to view details about a fictional *`my-example-widget`* resource but doesn't have the fictional `lookoutvision:`*`GetWidget`* permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 lookoutvision:GetWidget on resource: my-example-widget
```

In this case, the policy for the `mateojackson` user must be updated to allow access to the *`my-example-widget`* resource by using the `lookoutvision:`*`GetWidget`* action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to Lookout for Vision.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in Lookout for Vision. However, the action requires the service to have

permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I want to allow people outside of my AWS account to access my Lookout for Vision resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether Lookout for Vision supports these features, see How Amazon Lookout for Vision works with IAM.
- To learn how to provide access to your resources across AWS accounts that you own, see Providing access to an IAM user in another AWS account that you own in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see Providing access to AWS accounts owned by third parties in the *IAM User Guide*.
- To learn how to provide access through identity federation, see Providing access to externally authenticated users (identity federation) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.

Compliance validation for Amazon Lookout for Vision

Third-party auditors assess the security and compliance of Amazon Lookout for Vision as part of multiple AWS compliance programs. Amazon Lookout for Vision is compliant with General Data Protection Regulation (GDPR).

To learn whether an AWS service is within the scope of specific compliance programs, see AWS services in Scope by Compliance Program and choose the compliance program that you are interested in. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- Security Compliance & Governance – These solution implementation guides discuss architectural considerations and provide steps for deploying security and compliance features.
- HIPAA Eligible Services Reference – Lists HIPAA eligible services. Not all AWS services are HIPAA eligible.
- AWS Compliance Resources – This collection of workbooks and guides might apply to your industry and location.
- AWS Customer Compliance Guides – Understand the shared responsibility model through the lens of compliance. The guides summarize the best practices for securing AWS services and map the guidance to security controls across multiple frameworks (including National Institute of Standards and Technology (NIST), Payment Card Industry Security Standards Council (PCI), and International Organization for Standardization (ISO)).
- Evaluating Resources with Rules in the *AWS Config Developer Guide* – The AWS Config service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- AWS Security Hub – This AWS service provides a comprehensive view of your security state within AWS. Security Hub uses security controls to evaluate your AWS resources and to check your compliance against security industry standards and best practices. For a list of supported services and controls, see Security Hub controls reference.
- Amazon GuardDuty – This AWS service detects potential threats to your AWS accounts, workloads, containers, and data by monitoring your environment for suspicious and malicious activities. GuardDuty can help you address various compliance requirements, like PCI DSS, by meeting intrusion detection requirements mandated by certain compliance frameworks.
- AWS Audit Manager – This AWS service helps you continuously audit your AWS usage to simplify how you manage risk and compliance with regulations and industry standards.

Resilience in Amazon Lookout for Vision

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

Infrastructure security in Amazon Lookout for Vision

As a managed service, Amazon Lookout for Vision is protected by AWS global network security. For information about AWS security services and how AWS protects infrastructure, see AWS Cloud Security. To design your AWS environment using the best practices for infrastructure security, see Infrastructure Protection in *Security Pillar AWS Well-Architected Framework*.

You use AWS published API calls to access Lookout for Vision through the network. Clients must support the following:

- Transport Layer Security (TLS). We require TLS 1.2 and recommend TLS 1.3.
- Cipher suites with perfect forward secrecy (PFS) such as DHE (Ephemeral Diffie-Hellman) or ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.

Monitoring Amazon Lookout for Vision

Monitoring is an important part of maintaining the reliability, availability, and performance of Amazon Lookout for Vision and your other AWS solutions. AWS provides the following monitoring tools to watch Lookout for Vision, report when something is wrong, and take automatic actions when appropriate:

- *Amazon CloudWatch* monitors your AWS resources and and the applications you run on AWS in real time. You can collect and track metrics, create customized dashboards, and set alarms that notify you or take actions when a specified metric reaches a threshold that you specify. For example, you can have CloudWatch track CPU usage or other metrics of your Amazon EC2 instances and automatically launch new instances when needed. For more information, see the Amazon CloudWatch User Guide.
- *Amazon CloudWatch Logs* enables you to monitor, store, and access your log files from Amazon EC2 instances, CloudTrail, and other sources. CloudWatch Logs can monitor information in the log files and notify you when certain thresholds are met. You can also archive your log data in highly durable storage. For more information, see the Amazon CloudWatch Logs User Guide.
- *Amazon EventBridge* can be used to automate your AWS services and respond automatically to system events, such as application availability issues or resource changes. Events from AWS services are delivered to EventBridge in near real time. You can write simple rules to indicate which events are of interest to you and which automated actions to take when an event matches a rule. For more information, see Amazon EventBridge User Guide.
- *AWS CloudTrail* captures API calls and related events made by or on behalf of your AWS account and delivers the log files to an Amazon S3 bucket that you specify. You can identify which users and accounts called AWS, the source IP address from which the calls were made, and when the calls occurred. For more information, see the AWS CloudTrail User Guide.

Monitoring Lookout for Vision with Amazon CloudWatch

You can monitor Lookout for Vision using CloudWatch, which collects raw data and processes it into readable, near real-time metrics. These statistics are kept for 15 months, so that you can access historical information and gain a better perspective on how your web application or service is performing. You can also set alarms that watch for certain thresholds, and send notifications or take actions when those thresholds are met. For more information, see the Amazon CloudWatch User Guide.

The Lookout for Vision service reports the following metrics in the `AWS/LookoutVision` namespace.

Metric	Description
`DetectedAnomalyCount`	The number of anomalies detected in a project Valid Statistics: `Sum`, `Average` Unit: Count
`ProcessedImageCount`	The total number of images run through anomaly detection Valid Statistics: `Sum`, `Average` Unit: Count
`InvalidImageCount`	The number of images that were invalid and could not return a result Valid Statistics: `Sum`, `Average` Unit: Count
`SuccessfulRequestCount`	The number of successful API calls Valid Statistics: `Sum`, `Average` Unit: Count
`ErrorCount`	The number of API errors Valid Statistics: `Sum`, `Average` Unit: Count
`ThrottledCount`	The number of API errors that were due to throttlin g Valid Statistics: `Sum`, `Average`

Metric	Description
	Unit: Count
`Time`	The time in milliseconds for Lookout for Vision to compute the anomaly detection Valid Statistics: `Data Samples,Average` Units: Milliseconds for `Average` statistics and Count for `Data Samples` statistics
`MinInferenceUnits`	The minimum number of inference units specified during the `StartModel` request. Valid statistics: `Average` Unit: Count
`MaxInferenceUnits`	The maximum number of inference units specified during the `StartModel` request. Valid statistics: `Average` Unit: Count
`DesiredInferenceUnits`	The number of inference units to which Lookout for Vision is scaling up or down. Valid statistics: `Average` Unit: Count

Metric	Description
`InServiceInferenceUnits`	The number of inference units that the model is using. Valid statistics: `Average` It is recommended that you use the Average statistic to obtain the 1 minute average of how many instances are used. Unit: Count

The following dimensions are supported for the Lookout for Vision metrics.

Dimension	Description
`ProjectName`	You can split metrics by project to see which projects are having problems or need to be updated.
`ModelVersion`	You can split metrics by model version to see which models are having problems or need to be updated.

Logging Lookout for Vision API calls with AWS CloudTrail

Amazon Lookout for Vision is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in Lookout for Vision. CloudTrail captures all API calls for Lookout for Vision as events. The calls captured include calls from the Lookout for Vision console and code calls to the Lookout for Vision API operations. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for Lookout for Vision. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to Lookout for Vision, the IP address from which the request was made, who made the request, when it was made, and additional details.

To learn more about CloudTrail, see the AWS CloudTrail User Guide.

Lookout for Vision information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in Lookout for Vision, that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see Viewing Events with CloudTrail Event History.

For an ongoing record of events in your AWS account, including events for Lookout for Vision, create a trail. A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see the following:

- Overview for Creating a Trail
- CloudTrail Supported Services and Integrations
- Configuring Amazon SNS Notifications for CloudTrail
- Receiving CloudTrail Log Files from Multiple Regions and Receiving CloudTrail Log Files from Multiple Accounts

All Lookout for Vision actions are logged by CloudTrail and are documented in the Lookout for Vision API reference documentation. For example, calls to the `CreateProject`, `DetectAnomalies` and `StartModel` actions generate entries in the CloudTrail log files.

If you monitor Amazon Lookout for Vision API calls, you might see calls to the following APIs.

- lookoutvision:StartTriallDetection
- lookoutvision:ListTriallDetection
- lookoutvision:DescribeTrialDetection

These special calls are used by Amazon Lookout for Vision to support various operations related to trial detection. For more information, see Verifying your model with a trial detection task.

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or AWS Identity and Access Management user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see the CloudTrail userIdentity Element.

Understanding Lookout for Vision log file entries

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on. CloudTrail log files aren't an ordered stack trace of the public API calls, so they don't appear in any specific order.

The following example shows a CloudTrail log entry that demonstrates the `CreateDataset` action.

```
{
  "eventVersion": "1.08",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AROAYN4CJAYDEXAMPLE:user",
    "arn": "arn:aws:sts::123456789012:assumed-role/Admin/MyUser",
    "accountId": "123456789012",
    "accessKeyId": "ASIAYN4CJAYEXAMPLE",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AROAYN4CJAYDCGEXAMPLE",
        "arn": "arn:aws:iam::123456789012:role/Admin",
        "accountId": "123456789012",
        "userName": "Admin"
      },
      "webIdFederationData": {},
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2020-11-20T13:15:09Z"
      }
    }
```

```
  },
  "eventTime": "2020-11-20T13:15:43Z",
  "eventSource": "lookoutvision.amazonaws.com",
  "eventName": "CreateDataset",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "128.0.0.1",
  "userAgent": "aws-cli/3",
  "requestParameters": {
    "projectName": "P1",
    "datasetType": "train",
    "datasetSource": {
      "groundTruthManifest": {
        "s3Object": {
          "bucket": "myuser-bucketname",
          "key": "training.manifest"
        }
      }
    },
    "clientToken": "EXAMPLE-0526-47dd-a5d3-2ca975820a34"
  },
  "responseElements": {
    "status": "CREATE_IN_PROGRESS"
  },
  "requestID": "EXAMPLE-15e1-4bc9-be38-cda2537c75bf",
  "eventID": "EXAMPLE-c5e7-43e0-8449-8d9b87e15acb",
  "readOnly": false,
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "eventCategory": "Management",
  "recipientAccountId": "123456789012"
}
```

Creating Amazon Lookout for Vision resources with AWS CloudFormation

Amazon Lookout for Vision is integrated with AWS CloudFormation, a service that helps you model and set up your AWS resources so that you can spend less time creating and managing your resources and infrastructure. You create a template that describes all the AWS resources that you want, and AWS CloudFormation takes care of provisioning and configuring those resources for you.

You can use AWS CloudFormation to provision and configure Amazon Lookout for Vision projects.

When you use AWS CloudFormation, you can reuse your template to set up your Lookout for Vision projects consistently and repeatedly. Just describe your projects once, and then provision the same projects over and over in multiple AWS accounts and Regions.

Lookout for Vision and AWS CloudFormation templates

To provision and configure projects for Lookout for Vision and related services, you must understand AWS CloudFormation templates. Templates are formatted text files in JSON or YAML. These templates describe the resources that you want to provision in your AWS CloudFormation stacks. If you're unfamiliar with JSON or YAML, you can use AWS CloudFormation Designer to help you get started with AWS CloudFormation templates. For more information, see What is AWS CloudFormation Designer? in the *AWS CloudFormation User Guide*.

For reference information about Lookout for Vision projects, including examples of JSON and YAML templates, see LookoutVision resource type reference.

Learn more about AWS CloudFormation

To learn more about AWS CloudFormation, see the following resources:

- AWS CloudFormation
- AWS CloudFormation User Guide
- AWS CloudFormation API Reference
- AWS CloudFormation Command Line Interface User Guide

Access Amazon Lookout for Vision using an interface endpoint (AWS PrivateLink)

You can use AWS PrivateLink to create a private connection between your VPC and Amazon Lookout for Vision. You can access Lookout for Vision as if it were in your VPC, without the use of an internet gateway, NAT device, VPN connection, or AWS Direct Connect connection. Instances in your VPC don't need public IP addresses to access Lookout for Vision.

You establish this private connection by creating an *interface endpoint*, powered by AWS PrivateLink. We create an endpoint network interface in each subnet that you enable for the interface endpoint. These are requester-managed network interfaces that serve as the entry point for traffic destined for Lookout for Vision.

For more information, see Access AWS services through AWS PrivateLink in the *AWS PrivateLink Guide*.

Considerations for Lookout for Vision VPC endpoints

Before you set up an interface endpoint for Lookout for Vision, review Considerations in the *AWS PrivateLink Guide*.

Lookout for Vision supports making calls to all of its API actions through the interface endpoint.

VPC endpoint policies are not supported for Lookout for Vision. By default, full access to Lookout for Vision is allowed through the interface endpoint. Alternatively, you can associate a security group with the endpoint network interfaces to control traffic to Lookout for Vision through the interface endpoint.

Creating an interface VPC endpoint for Lookout for Vision

You can create an interface endpoint for Lookout for Vision using either the Amazon VPC console or the AWS Command Line Interface (AWS CLI). For more information, see Create an interface endpoint in the *AWS PrivateLink Guide*.

Create an interface endpoint for Lookout for Vision using the following service name:

```
com.amazonaws.region.lookoutvision
```

If you enable private DNS for the interface endpoint, you can make API requests to Lookout for Vision using its default Regional DNS name. For example, `lookoutvision.us-east-1.amazonaws.com`.

Creating a VPC endpoint policy for Lookout for Vision

An endpoint policy is an IAM resource that you can attach to an interface endpoint. The default endpoint policy allows full access to Lookout for Vision through the interface endpoint. To control the access allowed to Lookout for Vision from your VPC, attach a custom endpoint policy to the interface endpoint.

An endpoint policy specifies the following information:

- The principals that can perform actions (AWS accounts, IAM users, and IAM roles).
- The actions that can be performed.
- The resources on which the actions can be performed.

For more information, see Control access to services using endpoint policies in the *AWS PrivateLink Guide*.

Example: VPC endpoint policy for Lookout for Vision actions

The following is an example of a custom endpoint policy for Lookout for Vision. When you attach this policy to your interface endpoint, it specifies that all users who have access to the VPC interface endpoint are allowed to call the `DetectAnomalies` API operation for the Lookout for Vision model `myModel` associated with project `myProject`.

```
{
   "Statement":[
      {
         "Principal":"*",
         "Effect":"Allow",
         "Action":[
            "lookoutvision:DetectAnomalies"
         ],
         "Resource": "arn:aws:lookoutvision:us-west-2:123456789012:model/myProject/
myModel"
      }
   ]
```

```
}
```

Quotas in Amazon Lookout for Vision

The following tables describe the current quotas within Amazon Lookout for Vision. For information about quotas that can be changed, see AWS service quotas.

Model quotas

The following quotas apply to the testing, training, and functionality of a model.

Resource	Quota
Supported file format	PNG and JPEG image formats
Minimum image dimension of image file in an Amazon S3 bucket	64 pixels x 64 pixels
Maximum image dimension of image file in an Amazon S3 bucket	4096 pixels X 4096 pixels is the maximum. Smaller dimensions are able to upload faster.
Differing image dimensions of image files used in a project	All images in the dataset must have the same dimensions
Maximum file size for an image in an Amazon S3 bucket	8 MB
Lack of labels	Images must be labeled as normal or anomaly before training. Images without labels are ignored during training.
Minimum number of images labeled normal in training dataset	10 for a project with separate training and test datasets. 20 for project with a single dataset.
Minimum number of images labeled anomaly in a training dataset	0 for a project with separate training and test datasets. 10 for a project with a single dataset.
Maximum number of images in classification training dataset	16,000

Resource	Quota
Maximum number of images in a classification test dataset	4,000
Minimum number of images labeled normal in test dataset	10
Minimum number of images labeled anomaly in test dataset	10
Maximum number of images in an anomaly localization training dataset	8000
Maximum number of images in an anomaly localization test dataset	800
Maximum number of images in trial detection dataset	2,000
Maximum dataset manifest file size	1 GB
Maximum number of training datasets in a model	1
Maximum training time	24 hours
Maximum testing time	24 hours
Maximum number of anomaly labels in a project	100
Maximum number of anomaly labels on a mask image	20
Minimum number of images for an anomaly label. To count, the image must contain only one type of anomaly label.	20 for a single dataset project. 10 for each dataset in a project with separate training and test datasets.

Document history for Amazon Lookout for Vision

The following table describes important changes in each release of the *Amazon Lookout for Vision Developer Guide*. For notification about updates to this documentation, you can subscribe to an RSS feed.

- **Latest documentation update:** February 20th, 2023

Change	Description	Date
End of support notice	End of support notice: On October 31, 2025, AWS will discontinue support for Amazon Lookout for Vision. After October 31, 2025, you will no longer be able to access the Lookout for Vision console or Lookout for Vision resources. For more informati on, visit this blog post.	October 10, 2024
Added example Lambda function	Example showing how to find anomalies with an AWS Lambda function. For more information, see Finding anomalies with an AWS Lambda function.	February 20, 2023
Updated the IAM guidance for AWS WAF	Updated guide to align with the IAM best practices . For more information, see Security best practices in IAM.	February 8, 2023
Added dataset export example	Added Python example showing how to use the `ListDatasetEntries` operation to export the	December 2, 2022

	datasets from an Amazon Lookout for Vision project. For more information, see Exporting datasets from a project (SDK).	
Updated getting started topic	Updated getting started to show creating an image segmentation model with an example dataset. For more information, see Getting started with Amazon Lookout for Vision.	October 20, 2022
Added troubleshooting topic	Added model training troubleshooting topic. For more information, see Troubleshooting model training.	October 17, 2022
Added topic on using Amazon SageMaker AI Ground Truth jobs	Instead of labeling images yourself, you can use Amazon SageMaker AI Ground Truth jobs to label images for classification and image segmentation models. For more information, see Using an Amazon SageMaker AI Ground Truth job.	August 19, 2022

Amazon Lookout for Vision now provides anomaly localization.	You can create a segmentation model that finds the locations on an image where different types of anomalies (such as a scratch, dent, or tear) are present, the label of the anomaly and the size of the anomaly, For more information, see Running your trained Amazon Lookout for Vision model.	August 16, 2022
Amazon Lookout for Vision now provides CPU inference on edge devices.	Amazon Lookout for Vision models can now be deployed to run inference locally on an x86 compute platform running Linux with just a CPU, without needing a GPU accelerator. For more information, see CPU accelerator.	August 16, 2022
Amazon Lookout for Vision can now automatically scale inference units.	To help with spikes in demand, Amazon Lookout for Vision can now scale the number of inference units that your model uses. For more information, see Running your trained Amazon Lookout for Vision model.	August 16, 2022
Java examples added	The Amazon Lookout for Vision developer guide now includes Java examples. For more information, see Getting started with the AWS SDK.	May 2, 2022

General availability of model deployment to an edge device	Model deployment to an edge device managed by AWS IoT Greengrass Version 2 is now generally available. For more information, see Using your Amazon Lookout for Vision model on an edge device.	March 14, 2022
Updated console bucket information	Updated information on console bucket contents and alternative approaches to creating the console bucket. For more information, see Step 4: Create the console bucket.	March 7, 2022
Create a manifest file from a CSV file	You can now simplify the creation of a manifest file by using a script that reads classification information from a CSV file. For more information, see Creating a manifest file from a CSV file.	February 10, 2022
Preview release of model deployment to an edge device	The preview release of model deployment to an edge device managed by AWS IoT Greengrass Version 2 is now available. For more informati on, see Using your Amazon Lookout for Vision model on an edge device.	December 7, 2021

New Python and Java 2 examples added	Added Python and Java 2 examples for analyzing images with `DetectAnomalies` . For more information, see Detecting anomalies in an image.	September 7, 2021
New AWS managed policies added.	Amazon Lookout for Vision adds support for AWS managed policies. For more information, see AWS managed policies for Amazon Lookout for Vision.	May 11, 2021
Updated inference unit information.	Added information describing inference units and how they are charged. For more information, see Running your trained Amazon Lookout for Vision model.	March 15, 2021
General availability for Amazon Lookout for Vision.	Amazon Lookout for Vision is now generally available . Python code examples updated to handle asynchronous tasks such as training a model.	February 17, 2021
Tagging and AWS CloudFormation support added.	You can now tag Amazon Lookout for Vision models and create projects with AWS CloudFormation. For more information, see Tagging models and Creating Amazon Lookout for Vision projects with AWS CloudFormation.	January 31, 2021

New feature and guide	This is the initial release of the Amazon Lookout for Vision service *Amazon Lookout for Vision Developer Guide*.	December 1, 2020

AWS Glossary

For the latest AWS terminology, see the AWS glossary in the *AWS Glossary Reference*.